Computation of the Local Binary Pattern (LBP) descriptor of large scale images

October 17, 2014

Joshua Chen
jyc28@uclive.ac.nz
Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand

Supervisor: Professor Ramarkishnan Mukundan
mukundan@canterbury.ac.nz

Abstract

Local binary patterns (LBPs) are powerful texture descriptors that have recently found several applications in medical image analysis. Research in this field is currently directed towards parallel implementations suitable for processing large scale images. Programming tool-sets such as the Open Computing Language (OpenCL) opened up opportunities for the development of various parallel algorithms and applications for General-Purpose GPU (GPGPU), all executable across heterogeneous OpenCL compliant platforms. In this report, we give an introduction to the LBP texture descriptor and the OpenCL framework. We will also discuss the computation of LBP descriptors for high resolution tissue images of biopsy samples, and outline various implementation aspects of the algorithm in OpenCL. Experiments were conducted on consumer-grade graphical processing units (GPUs) and a central processing unit (CPU), addressing relations to more than algorithmic complexities but limits on physical resources too.

Acknowledgments

Family, close family friends, and friends made during these four years in university for their continual support and encouragement.
Dr Mukundan, for his advice, support and guidance as my supervisor.

Contents

1 Introduction

1.1 Motivation and Background

In this day and age, large resolution and high magnification digital microscopy images of biopsy samples are commonly used to grade and stage cancerous tumours. Leaders in this field are focused in providing patients and physicians with extensive and detailed information not only for detection, but also to obtain the exact location, volume, stage and grade of such tumours. Such reliable tumour identification precision can now be commonly found in many automatic cancer detection systems. Furthermore, these systems relies on new generation medical equipments built with improved image processing and display capabilities that markedly enhance image color and resolution. Processing these giga-byte order images to identify tissue characteristics and other cytological features could take enormous periods of computational time. Therefore these images can be first subdivided into many smaller segments and later processed using parallel implementations of algorithms.

This work consisted of looking into why Local Binary Patterns (LBP) is a great algorithm for classification, and why parallel implementations of it should be done with the OpenCL framework. The basic LBP was implemented in OpenCL and tested with various graphics processing units (GPUs). Measures and best practices were taken to optimize performance within OpenCL; they would be discussed in following chapters. One of OpenCL's primary advantages is its heterogeneous programming environment, allowing it to be executed on many other types of processors and not only GPUs. Therefore our developed methods can be readily used on large parallel architectures such as the University of Canterbury Blue-Fern super computer.

The GPU was the primary choice of processor to be tested on because it is well-suited to address problems that can be expressed as data-parallel computations, like LBP. The LBP operator is like a single program fed with multiple sets of data elements, one at a time thus its potential to be performed parallely. Compared to the CPU, the GPU devotes more transistors to data processing rather than data caching and flow control. And because the LBP program is executed only once for every set of data elements, it has a low requirement for sophisticated flow control. Furthermore GPU excels in compute intensive, highly parallel computations, allowing LBP to be executed on many sets of data with high arithmetic intensity. This conceals memory access latency with calculations instead of big data caches.

1.2 A Brief History of Image Processing

Visual detection and classification are important and complex aspects in image processing. A major problem is that real world image textures are often not uniform due to many different variations. This variability can range from lighting conditions and clutter in backgrounds to image acquisition which itself is not always consistent due to noise and focus instability. Also, in order to produce automated

detection and texture classification, quality descriptors and classifiers are needed. Over the past few years, progress in machine learning has led to methods that are adaptable to variability in images. However, in practice there are too many constraints on the type of input required for quality results. This shows how crucially good descriptors are needed and in demand.

The texture of images refers to the appearance, structure and arrangement of the parts of an object within the image. Images used for diagnostic purposes in clinical practice are digital and mostly two-dimensional. Texture analysis has been researched since the 1960s; in principle it is a technique for evaluating the position and intensity of signal features, that is, pixels, and their gray level intensities. The distribution of these pixels can be computed to produce mathematical parameters which characterize the texture type and thus the underlying structure of the objects shown in the image; these values are also known as texture features.

The use of whole feature distributions in texture classification instead of single values texture measures such as means and variances was very rare in the early 1990s. However, during that time studies showed much important information contained in the distributions of feature values were lost through the usage of these single texture measures. Furthermore these texture classification methods were found to assume, either explicitly or implicitly, that the unknown samples to be classified are always identical to the training samples with respect to scale, orientation and gray-scale properties. Real-world textures are not like that, they are unpredictably subjected to varying illumination conditions and arbitrary spatial rotations constantly. This showed how unreliable past texture classifications were and their incompetence in handling real world images. Not to mention, the degree of computational complexity in their algorithms is too high [1] . A very helpful suggestion for future research from then was to develop texture measures which incorporate invariance to real-world factors such as orientation and scale, and can be classified with a low-computational complexity. From that, LBP was developed.

1.3 Local Binary Patterns (LBP)

A starting point for the research on Local Binary Patterns (LBP) was the idea that two-dimensional textures can be described by two complementary local measures: spatial structure (pattern) and contrast (the amount of local image texture). In terms of gray-scale and rotation invariance, these two properties are an interesting pair; spatial pattern is affected by rotation of the texture but contrast is not and contrast is affected by the gray-scale though spatial pattern is not. Therefore if a texture descriptor is capable in separating the texture's pattern information from its contrast, then invariance to monotonic gray scale changes can be obtained.

LBP was first published as part of a comparative study of texture operators in the International Conference on Pattern Recognition conference (ICPR 1994) [2], and an extended version of it in Pattern Recognition journal [3]. The relation of LBP to the texture spectrum method proposed by Wang and He [4] was found during the writing of the first paper on LBP. Years later it was also found that LBP developed for texture analysis is very similar to the census transform that was proposed at around the same time as LBP for computing visual correspondences in stereo matching [5].

Using distributions of LBP provided excellent texture classification; this was shown when it was utilized for unsupervised texture segmentation [6], obtaining results which were clearly better than

the state of the art during that time. It was further revised to handle textures on larger scales by using a simple multi-scale extension [7]. Local binary texture patterns were found to be uniform, corresponding to primitive micro-features, such as edges, corners, and spots; hence they can be regarded as feature detectors that are triggered by the best matching pattern. This presented LBP to be fundamental in the development of a generalized gray-scale and rotation invariant operator [8]. LBP was also combined with other descriptors such as SIFT, to consolidate their strengths, which proved to be effective in interest regions detection [9]. Due to its computational simplicity, LBP was used early in some applications such as visual inspection [10]. The high potential of LBP is clearly demonstrated from all these outcomes and motivated future research. LBP also happens to be one of the most used texture descriptors in medical image analysis, shown to be useful in describing medical images accurately in high detail. Much research lead to the development of variants of the LBP which are widely considered the state of the art among known texture descriptors [11].

The histogram is one of the most commonly used texture parameters for analysis of medical images [12]. Coming from the statistical class of texture techniques, an image histogram displays the count of pixels in an image possessing a given gray level value. Many other parameters can also be derived from the histogram, such as its mean, variance and percentiles.

The discrete occurrence histogram of the local binary texture patterns computed over an image or region of image was and still is a very powerful texture feature [8]. By computing the occurrence histogram, one can effectively combine the structural and statistical approaches. This was demonstrated this through LBP's detection of micro-structures such as edges, lines and spots, whose underlying distribution was estimated by an occurrence histogram. A later investigation was also carried out to find the relationship of LBP to a method based on multidimensional gray scale difference histograms [13]. The results of this research created a simplification of the LBP operator based on signed gray level differences. Vector quantization was used to reduce the dimensionality of the feature space of multidimensional histograms forming a one-dimensional texton histogram. Other researchers further found that instead of using only the LBP histogram derived from standard position textures, they could improve the discriminative classification power by extracting a certain small subset of LBPs from the histogram. This showed optimization in classifying tilted 3-D textures even without pre-processing [14].

LBP histograms are continually used in other fields than image analysis. In motion analysis, each pixel is modelled as a group of adaptive local binary pattern histograms that are calculated over a circular region around the pixel. The method was shown to be tolerant to illumination variations, the multi-modality of the background, and the introduction or removal of background objects [15]. In 3D textured surfaces, multiple LBP histograms were used as object models and this produced excellent results [16]. LBP histograms were also used in face analysis research, it enabled methods to be robust to face misalignments and pose variations.

1.4 General Purpose Graphics Processing Unit (GP-GPU)

The graphics processing units (GPU) on today's commodity video cards were originally designed for efficient data-parallel graphics computations, such as scene rasterization and lighting effects. However as they evolve into extremely powerful and flexible processors, the architecture's huge memory bandwidth and computational processing power gained tremendous popularity among researchers and developers and still is. Increasing parallelism rather than clock rate, has become the primary engine of processor performance growth, and this trend is likely to continue. Researchers have found that

exploiting the GPU can accelerate some problems by over an order of magnitude over the CPU. Early attempts to exploit GPUs high level of parallelism for applications beyond the domain of graphics required developers first recast their problem into graphic primitives using high-level shading languages such as Cg, HLSL, GLSL, and reinterpret graphical results. However recent high level toolkits like Compute Unified Device Architecture (CUDA) [17, 18] from NVIDIA and frameworks like Open Computing Language (OpenCL) [19] have made this computational power easily accessible by enabling developers to write functions called kernels that execute in parallel on the streaming processors, without mastering graphic terms [20]. This effort in harnessing power for general-purpose computing is collectively known as General-Purpose computing on Graphics Processing Units (GPGPU).

1.5 Open Computing Language (OpenCL)

However every programming framework has its unique method for application development, such as CUDA which is a proprietary API and set of language extensions that only works on NVIDIAs GPU. This can be inconvenient as software development must be rebuilt from scratch each time a new platform is brought into the market [21]. Therefore OpenCL, by the Khronos Group, is a framework that allows parallel programming across heterogeneous systems, such as Central Processing Units (CPU), GPUs, Digital Signal Processors (DSP), and many other types of processors. Due to its portability, it is questionable whether OpenCLs performance is compromised. Some of recent research showed that CUDA performed better in memory transfers of data to and fro the GPU, and is a better choice for high performance [22]. However there were investigations that those results could be due to unfair comparisons [23, 24]. Finally it was shown that OpenCLs portability does not hugely affect its performance, proving it to be an compatible alternative to CUDA. Unlike a CUDA kernel, an OpenCL kernel can be compiled at runtime, which would add to an OpenCLs run time. However this just-in-time compile may allow the compiler to generate code that makes better use of the GPU.

1.6 Summary

Being designed to be massively computed in parallel, the Local Binary pattern operator is well suited to be implemented on the likewise concurrent architecture of a GPU. Since every LBP pattern is independent of each other, any amount of patterns can be computed simultaneously. Building these parallel programs with OpenCL is relatively straightforward as there is no need to learn device specific languages; code can be written once and run on any OpenCL-compliant hardware. It is also a bonus that high-speed image processing is one of OpenCL's most important strengths too.

1.7 Overview of the Report

In the Related Work chapter, there were four papers written that are closely related to this work. They were reviewed and their similarities explained in their respective sections.
Design and Implementation consists of many components; firstly a detailed description of the LBP methodology is provided. Followed by a more concrete introduction to the OpenCL framework with comparisons provided to the NVIDIA CUDA architecture. The CUDA^{TM} architecture is described here because the GPUs we ran our implementation tests on are NVIDIA GPUs. CUDA was designed to support application programming interfaces like OpenCL therefore it would be helpful to understand the relations between the two and this would be crucial to optimize the performance of any NVIDIA GPU. Next we consider the many concepts of OpenCl that were fundamental to the development of parallelized LBP applications, before going into detail in each different implementation.
In Evaluation, the various GPUs and CPUs used are mentioned, along with the various profiling tool-

kits and their roles in testing the applications. Finally a full compiled analysis is shown and explained in depth.
Lastly we have our conclusion and suggested future work that can be done.

2 Related Work

2.1 Real-Time face Detection on GPU using OpenCL

GPU has an inherent parallel execution architecture and higher computing capacity compared to the CPU. In this work a parallelized variant of LBP was implemented on the GPU using OpenCL with a focus on real time face detection [25]. It required using OpenCV to capture an image of a face before converting it to grey-scale, performing LBP operations on it then extracting the LBP features and consolidating them together into a histogram. Face detection was done through chi-square distance classification on local binary pattern histograms. For parallel processing of the algorithm, the image is subdivided into 16x16 pixel blocks, designed such that each work-item processes a block. After processing, each work-item stores the LBP values of their block into a 18x18 matrix. All the matrices are combined to form the final LBP histogram. Though it was not mentioned, this approach is called Map-reduce. Much of the computation is done in the GPU, only the input image and the final histogram are transferred between the CPU and GPU, therefore overheads associated with data transfer is minimal.

Input Resolution	640x480
Sub Histograms	256
CPU(i5 3rd generation)	109 ms
AMD(7670M)	20 ms

Table 2.1: Performance of Implementation [25]

2.2 Parallel Implementation of LBP based Face Recognition on GPU using OpenCL

This paper was the predecessor of the one above [26]. 16x16 pixel windows were also used here. The implementation consisted of three steps: 1) Generation of the LBP image 2)Feature histogram generation 3)Computation of the distance for classification. Every image 8-bit pixel was accessed through work items within work-groups; synchronized using events and barriers. The output LBP image was then updated pixel by pixel independently from each work item. The LBP image is split into image blocks; a 256 bins local histogram to be generated from each block. It was emphasized for every work group to take information from their block and summarize their local histogram data in the local memory of the device. They claimed that the memory over heads can be reduced by having many work items contribute to a locally shared memory. Also if the number of work groups was increased, large amount of local data would be transferred to the global memory which leads to more memory overloads and a longer execution time. Furthermore they stated that it was efficient to use local memory area to increase the number of groups. There were definitely some confusion understanding their LBP histogram generating implementation and it was not very clear too. The 16 by 16 pixel

windows used which I assumed were the size of work-groups also did not correlate to the written number of sub-histograms of various image sizes in their presented performance table. Moreover it was also suggested to have the number of work-groups as close to the number of computational units or SIMD cores in the GPU device; this was found not to be give any benefits in our work. The sizes of images used here are only from 128x128 to the largest being 1024x1024.

Method	*Sub Histograms*	Recognition Performance	
		GPU(%)	*CPU(%)*
Non-Weighted $LBP^{u2}_{(8,1)}$	8	74.3	74.25
	16	78.5	77.25
Non-Weighted $LBP^{u2}_{(8,2)}$	8	75	77.5
	16	78.25	79.75
Weighted $LBP^{u2}_{(8,1)}$	8	74.75	74.5
	16	79	77.75
Weighted $LBP^{u2}_{(8,1)}$	8	78.25	78.75
	16	78.75	79

Table 2.2: Performance Efficiency of Algorithm [26]

Image Size	*Sub Histograms*	CPU (Core 2 Duo)	GPU (AMP 6500)
		Feature Ext.(ms)	*Feature Ext.(ms)*
128x128	16	34.5	25.9
225x225	49	105.3	27.3
256x256	64	134.6	28.5
512x512	256	545.6	36.3
1024x1024	1024	2204.7	64.7

Table 2.3: Performance of LBP Feature Extraction [26]

2.3 Face Detection with Improved Local Binary Patterns in CUDA

Here improved local binary patterns were used which is a variation of LBP with weights [27]. The work-group blocks were sizes of from 24x24 up to maximum 32x32 pixels. Their image extraction was different from ours such that square patches in various positions and sizes within the image were considered. A face classifier was trained using images only of size 24x24 pixels, therefore each extracted patch was resized to 24x24 pixels or maximum 32x32 pixels. The partitioning of work-items/threads was structured with a 2D grid where each row corresponds to a patch and a column corresponds to an ILBP feature at a particular location. This enabled them to process patches without

additional memory mapping; but this limits the grid size to maximum 65,535 blocks, that is, approximately an input image size of 300x300 pixels. These 24x24 patches were saved in global memory; shared memory was also used to further decrease calculation latencies by saving each 3x3 neighbourhood into shared memory prior to performing calculations. They had two GPU implementations, one with only global memory and the other with shared memory. GPU performance with shared memory was shown to be quicker; details of the difference in implementation between both were not elaborated on much.

2.4 Local Binary Pattern based Texture Analysis in Real Time using a Graphics Processing Unit

This work proposed an reformulation of the multi-scaled Local Binary patterns texture operator on a consumer-grade graphical processing unit (GPU), yielding a 14 to 18-fold run time reduction compared to standard CPU implementation [28]. Multi-scale and uniform extensions of the local binary pattern were used to improve results for texture classification tasks. Here neither OpenCL nor CUDA were used. Implementation was done through GPU fragment shader programs using NVIDIAs Cg framework/programming language. Each shader program takes an input image, alters it then outputs it as input for the next program. The pipeline processing involved, uploading the grey-scale input image, spreading it over all the color channels, blurring using two steps of Gaussian filter kernels, then the computation of LBP before mapping the pixels to an output image. Figure 2.1 shows this. Vectorization was done to further improve the implementation. Images ranging with dimension 512x512 to 1024x1024 could be used as input. Implementation was based on 32-bit integer computation and 16-bit floating point accuracy to achieve fast processing speeds. About 0.5 percent of the computed LBP values were inaccurate. Though not much can be taken from here as OpenCL was not used but separate programs put together, we could see the huge performance leaps from the CPU, how LBP is a great method suited for parallelization in many ways. For the experiment 1024x1024 images were used only.

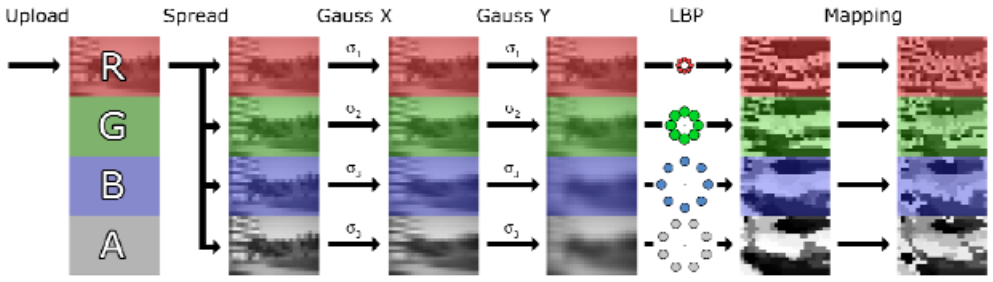


Figure 2.1: Sequence of GPU shader Programs for LBP Image Computation [28]

CPU / GPU	Total Processing Time	Upload	Computation	Download
Core2Quad 2.4GHz	**1150 ms** (1 core used)	-	1150 ms	-
GeForce 7600 GT	83 ms	17 ms	20 ms	46 ms
GeForce 8600 GT	72 ms	17 ms	11 ms	44 ms
GeForce 8800 GTS	65 ms	17 ms	5 ms	43 ms

Table 2.4: Speed Comparisons CPU/GPUs [28]

3 Design and Implementation

In this section, we will be looking into the methodology of LBP; its two stages: generating the LBP image and LBP histogram. Then an overview of heterogeneous computing with OpenCL, its conceptual foundations and framework. Important capabilities and advantages of OpenCL will be discussed too.

3.1 Local Binary Patterns Methodology

Local binary pattern is a computationally efficient texture descriptor, transforming an image under examination into a compact image of integer labels describing at least one essential characteristic within the small-scale appearances of the image. These locally generated LBP labels are encoded into binary codes and processed to achieve a unique representation of the image invariant to changes in illumination intensities. These LBP labels are then extracted and organized into a histogram that is used for further image analysis.There are many ways to generate this histogram; one way would be to concatenate the local histograms from many small regions divided equally in the image.

A key advantage of the LBP histogram is its normalization for translation. Rotations of a textured input image can cause the LBP patterns to translate into a different location and to rotate about their origin; computing the histogram of the LBP labels normalizes the translation.

The original version of the local binary pattern operator works in a 3x3 pixel window of an image. The neighbouring pixels in this window are thresholded by the center pixel intensity value, multiplied by powers of two and then summed to obtain the label for that center pixel. As the 3x3 neighbourhood consists of 8 pixels, there is a total of $2^8 = 256$ different labels that can be obtained.

Consider an image I(x,y) and let g_p denote the gray value of a sampling point with coordinates X_p, Y_p in an evenly spaced circular neighborhood of P sampling points and radius R around point x_c, y_c: [15]

$$g_p = I(x_p, y_p), \quad p = 0, \ldots, P-1 \quad \textit{and} \tag{3.1}$$

$$x_p = x_c + R\cos(2\pi p/P), \tag{3.2}$$

$$y_p = y_c - R\sin(2\pi p/P). \tag{3.3}$$

Assume that the local texture T of the image I(x,y) is characterised by the joint distribution of gray values of P+1 (P>0) pixels. Moreover, let g_c denote the gray level of the local texture neighbourhood centre pixel x_c, y_c) i.e. $g_c = I(x_c, y_c)$:

$$T = t(g_c, g_0, g_1, \ldots, g_{P-1}) \tag{3.4}$$

Without loss of information, the center pixel value can be subtracted from the neighbourhood:

$$T = t(g_c, g_0 - g_c, g_1 - g_c, \ldots, g_{P-1} - g_c) \tag{3.5}$$

The signs of the differences are considered:

$$t(s(g_0 - g_c), s(g_1 - g_c, \ldots, s(g_{P-1} - g_c)), \tag{3.6}$$

where s(z) is the thresholding step functions

$$s(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0. \end{cases} \tag{3.7}$$

Next by summing the thresholded differences, weighted by powers of two, the LBP_P,R operator is defined:

$$LBP_P,R(x_c, y_c) = \sum_{p=0}^{P-1} s(g_p - g_c)2^p \tag{3.8}$$

The signs of the differences in a local texture neighbourhood are interpreted as a P-bit binary number, resulting in 2^p distinct values for the LBP code. The local gray-scale distribution, i.e. texture, can thus be approximately described with a 2^p bin discrete distribution of LBP label codes. For center pixels that are on the borders, its neighbouring pixels outside of the image can be found by wrapping around the image.

To create an LBP representation of an input texture image, it must first be converted to grey-scale before this operator is applied to each individual pixel within the image as shown in Figure 3.3 . Figure 3.2 illustrates the algorithm of the basic LBP operator, by producing an LBP code for very pixel. As was mentioned before, the neighbourhood surrounding a pixel consists of eight pixels, giving a total of 256 different possible labels depending on the relative gray values of that center pixel and the pixels in the neighbourhood. The contrast measure is obtained by subtracting the average gray level below the center pixel from that of above or equal to the center pixel's gray level value. If all eight thresholded neighbours of the center pixel have the same value (0 or 1), the value of contrast is set to zero. The distributions of the LBP codes can be represented as a histogram to be used as features in classification or segmentation. See the histogram in Figure 3.3 .

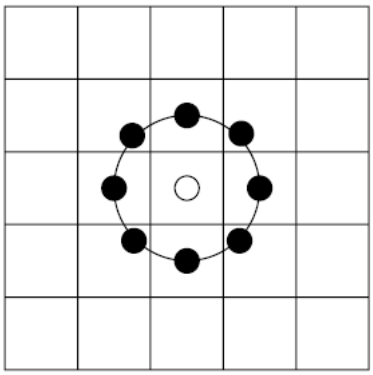


Figure 3.1: The circular (8,1) neighbourhood of the basic LBP operator

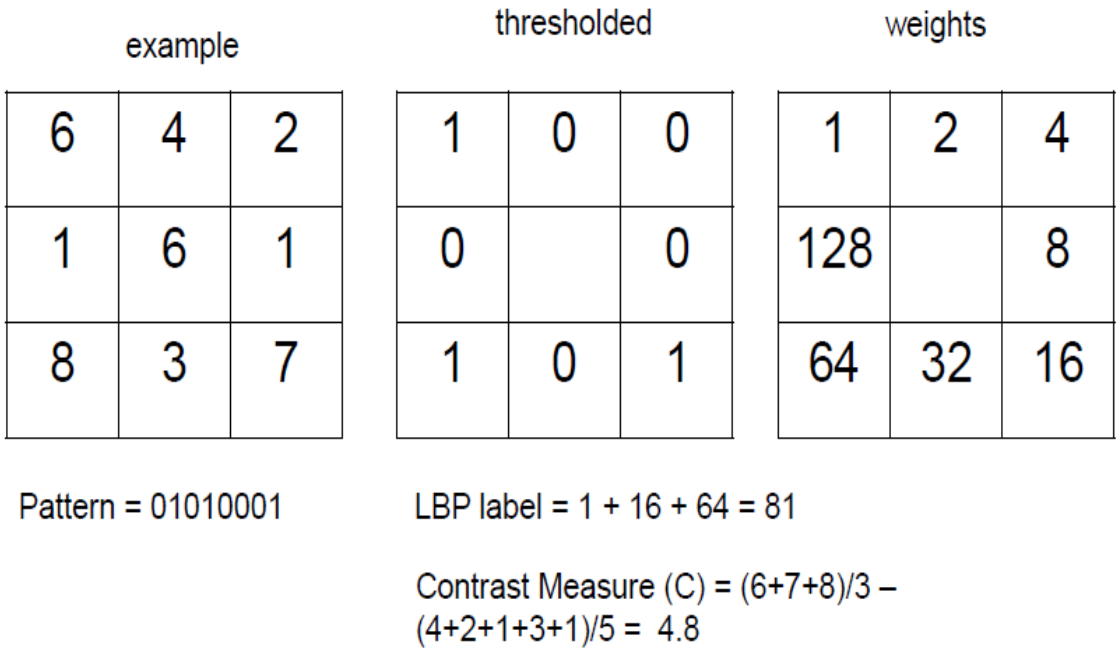


Figure 3.2: Example of applying the basic LBP operator

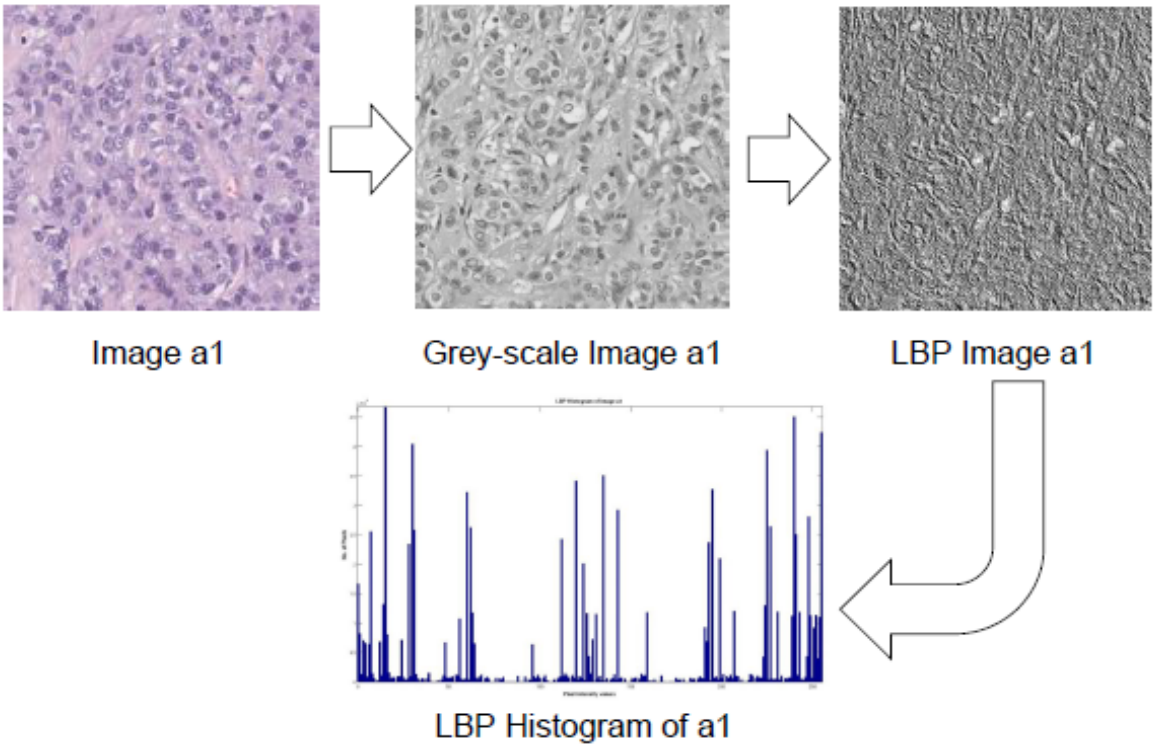


Figure 3.3: Processing pipeline of LBP

3.2 Compute Unified Device Architecture (CUDA)

In late 2006, NVIDIA introduced their general purpose parallel computing architecture, CUDA^{TM} that leverage the parallel compute engine in NVIDIA GPUs to efficiently solve many complex problems [17]. Figure 3.4 shows the compatible languages and application programming interfaces that can be used to program the CUDA architecture.

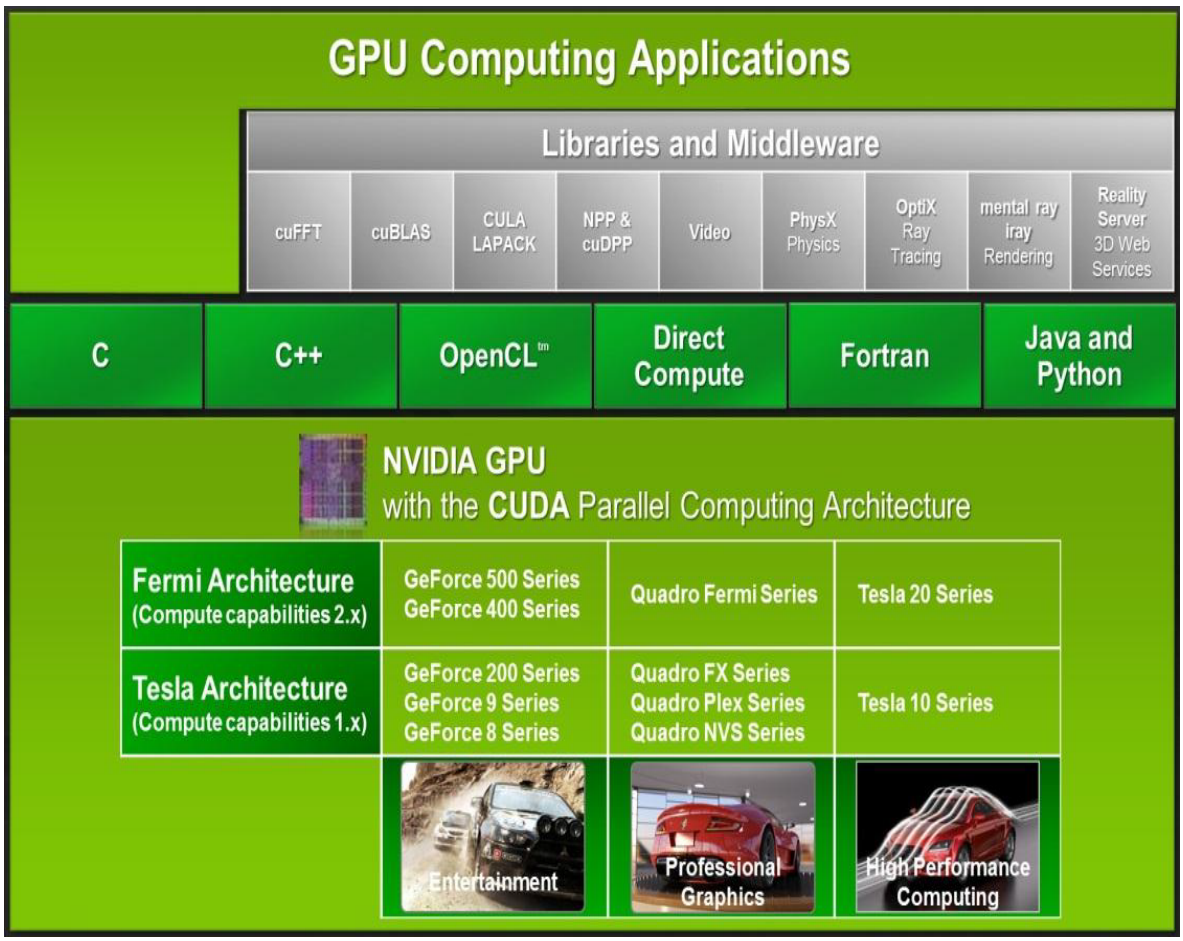


Figure 3.4: Various languages and Application Programming Interfaces supported by CUDA [17]

CUDA's programming model is designed with three key characteristics: a hierarchy of thread groups, a hierarchy of shared memories, and barrier synchronization. The CUDA architecture is very similar to the OpenCL architecture. Therefore there would be some relations made to CUDA as components of OpenCL are being explained in the next section. But before that, some terminology that CUDA and OpenCL uses that represents the same abstraction [17]: An OpenCL compute device, also known as a CUDA-enabled GPU, is built around an array of multi-threaded streaming multi-processors (SMs). A SM corresponds to an OpenCL compute unit; and within it consists of many OpenCL processing elements which are CUDA streaming processors. Refer to Figure 3.5, text in the green bubble are in CUDA terms and the normal white text are in OpenCL terms.

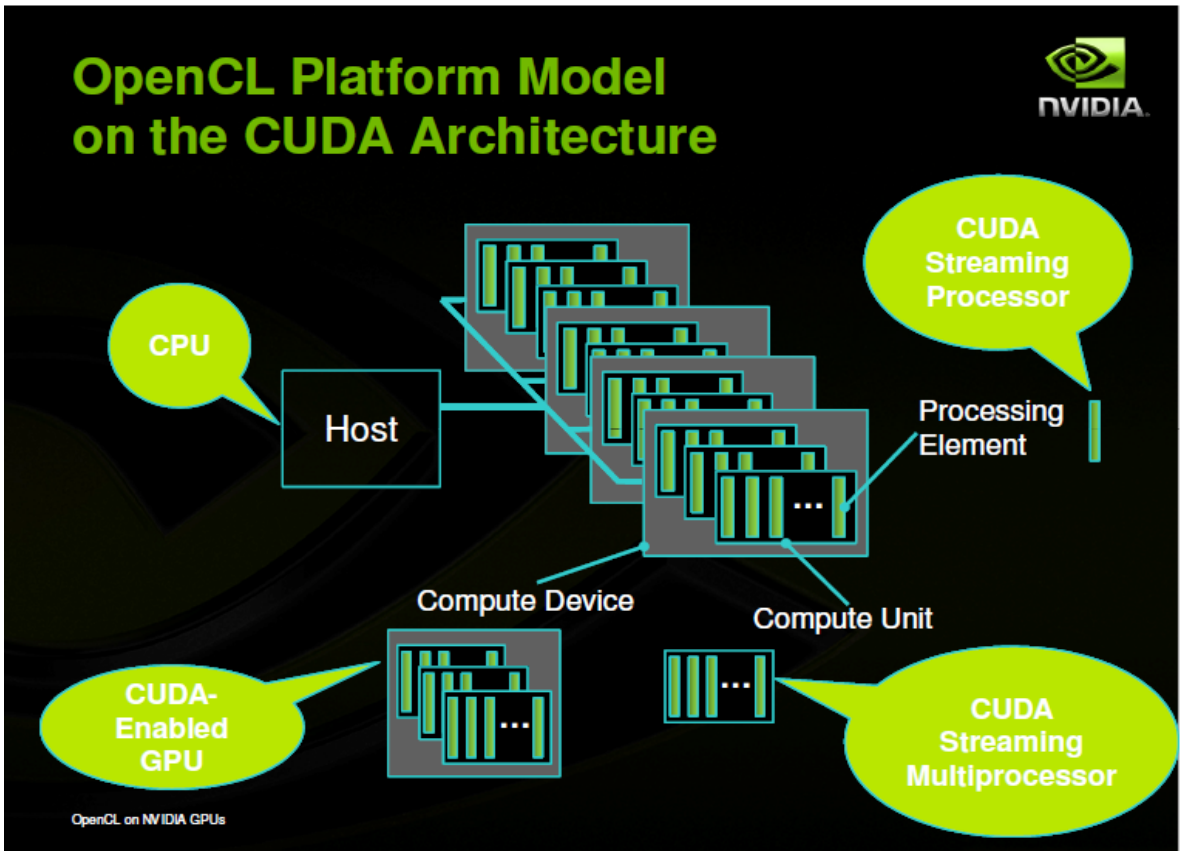


Figure 3.5: OpenCL Platform Model on CUDA Architecture
[29]

A SM executes an OpenCL work-item for a CUDA thread and OpenCL work-group for each CUDA thread block. A kernel is executed over an OpenCL NDRange by a grid of CUDA thread blocks. Each of these thread blocks that execute a kernel is uniquely identified by its work-group ID, and each thread by its global ID or by a combination of its local ID and work-group ID.

3.3 Open Computing Language (OpenCL) Framework

There are a series of APIs inside of OpenCL plus a programming environment for the kernels; giving the following models: Platform model, Execution model, Memory model, and Programming model. Every model was crucial to get our OpenCL LBP program functioning, though not all capabilities were needed. Below is a description of each model and related information to the implementation of LBP on OpenCL. Note that OpenCL and CUDA terminology will be both indicated when used.

3.3.1 Platform Model

This is a high level representation of any heterogeneous OpenCL-compliant platform. An OpenCL platform always includes a single **host** which is the CPU;and it handles all I/O and interaction with the user's program. The host is connected to one or more OpenCL **compute devices**. It can be a CPU, a GPU, a DSP, or any other processor provided by the hardware and supported by the OpenCL vendor.

3.3.2 Execution Model

The execution model defines an OpenCL application consisting of the host program and a collection of kernels to execute. The computational work of an OpenCL application takes place on the OpenCL devices. The host plays a very important role in the OpenCL application. It is on the host where the kernels are defined. When the host program issues the command that submits a kernel for execution on a compute device, the OpenCL runtime system creates an integer index space. An instance of an executing kernel is a **work-item**, and is identified from other work-items by its coordinates in the local index space, that is, its local ID.

Work items are organized into **work-groups** which exactly spans the global index space. Each work-group has its own unique work-group ID.The local index space is the index space inside a work-group. All work-groups are all of the same size, divided evenly from the global index space size. Work items are assigned a unique local ID within their own work-group, and a global ID to differentiate from other work-items. This is so that we can control the concurrency in OpenCL by manipulating how the work-items in a given work-group execute concurrently on the processing elements of a single compute unit. Work-items corresponds to processing elements, just as work-groups to compute units. An implementation may serialize the execution of kernels and even work- groups in a single kernel invocation; OpenCl only assures that the work-items within a work-group execute parallelly.

The index space spans an N-dimensioned range of values and is called an **NDRange**. Inside an OpenCL program, an NDRange is defined by an integer array of length N specifying the size of index space in each dimension. Usually the index space starts with a zero in each dimension.

The **context** for the OpenCL application is defined by the host, and is the environment where kernels are defined and executed. It consists of a collection of OpenCL devices used by the host, the OpenCL functions to run on the devices, program source code and executables, and lastly a set of memory objects visible to OpenCL devices for instances of an executed kernel. Not many modifications were done to the context in our OpenCL LBP program.

Command queues provide the interaction between the host and OpenCL devices through commands. Three types of commands are supported: kernel execution commands, memory transfer commands and synchronization commands. In a program, the context, command-queues, memory and program objects are first defined. Then memory objects are moved from the host onto devices by being filled with kernel arguments and then submitted to the command-queue for execution. When the kernel completes, output memory objects can be copied back to the host. These commands wait in the command-queue until they are executed on the OpenCl device. They launch in the order in which they appear in the command-queue and complete in that order too. Each device has its own one and only one command-queue. Only the first two command types were utilised in our OpenCL LBP program as there were not many commands to execute for them to need any modifications in synchronization.

The OpenCL execution model supports a wide range of programming models, but the ones used in our program were **task parallelism** and **data parallelism**.

3.3.2.1 Relation to CUDA

When an OpenCL program on the host invokes a kernel, the work-groups are distributed as thread blocks to the SMs. The threads of a thread block execute parallely on on SM. As the thread blocks finish, new blocks are launched on the vacant SMs. To manage an enormous amount of threads executing concurrently, a SM uses a unique architecture called SIMT (Single-instruction, Multiple-thread) that is described in Section OPTIMIZATION). This enables the SM to have fast barrier synchronization (will be described in the Programming model of OpenCL) with efficient thread creation and zero thread scheduling overheads, supporting very fine-grained parallelism.

3.3.3 Memory Model

OpenCL has two types of memory objects: **buffer objects** and **image objects**. A buffer object is a contiguous block of memory made available to kernels. Any type of data structure (subject to limitations of the OpenCL kernel programming language) can be mapped onto this buffer and accessed through pointers. Image objects, on the other hand, only holds images. The OpenCL framework provides functions to manipulate images, however other than these, the contents of image objects are hidden from the kernel.

The memory model also defines five distinctive memory regions: **Host memory**, **Global memory**, **Constant memory**, **Local memory** and **Private memory**. Host memory, as the name implies, is visible only to the host. Global memory permits read/write access to all work-items in all work-groups. The OpenCL device memory works with the host to support global memory. Constant memory is a region of global memory that remains constant during the execution of a kernel. Work-items have read-only access to these objects. Local memory is local to a work-group which runs on a compute unit. It is shared by all work-items in a work-group and can be used to allocate local variables. Processing elements have small regions of memory, providing work-items their own private memory.

When concurrent execution is involved, the memory model needs to carefully define how memory objects consistently interact in time with the kernel and host. Because local memory is shared only within a work-group, this is sufficient to define the memory consistency for local memory regions. For work-items within a group, global memory can be made consistent at a **work-group barrier**. However though this memory is shared between work-groups, there is no way to enforce consistency of global memory between different work-groups executing a kernel. For memory objects, OpenCL defines their loads and stores to behave in a different order for different work-items at any given moment. Therefore values seen in memory by a particular work-item may not be consistent with other work-items at all times. However when all work-items associated with a kernel complete, it is made sure the loads and stores for the memory objects are done before the kernel command is signalled as finished.

3.3.3.1 Relation to CUDA

Each multiprocessor has on-chip memory of the four following types:

1. A set of 32-bit registers

2. A parallel data cache or shared memory that is shared by all streaming processors and is where the OpenCL local memory resides

3. A read-only constant cache that is shared by all steaming processors and speeds up reads from OpenCl constant memory

4. A read-only texture cache that is shared by all streaming processors and speeds up reads from OpenCL image objects. The multiprocessor accesses the texture cache via a texture unit that is configured by OpenCL sampler objects; the region of device memory addressed by image objects is commonly referred to as a texture memory.

The number of blocks a multiprocessor can process at once referred to as the number of **active blocks** per multiprocessor depends on the required number of registers per thread and shared memory per block for a given kernel since all of the multiprocessors' registers and shared memory are divided among all threads of the active blocks. If there are not enough available registers or shared memory per multiprocessor to process at least one block, the kernel will not be executed.

3.3.4 Programming Model

The programming model is crucial in the planning towards the concurrency development of an OpenCL program. OpenCL was defined with two different programming models in consideration: **task parallelism** and **data parallelism**. Problems suited to data-parallel programming are arranged around data structures, the elements of which are updated concurrently. The key is **NDRange** of OpenCLs execution model. The data structures has to be aligned with the NDRange index space and mapped to OpenCL memory objects. Sequence of instructions to be applied concurrently are defined as work-items by the executing kernel.

Shared memory supports work-items in work-groups to share data. Care must be taken that regardless of the order work-items completes, the same results are reproduced. This requires synchronization between executions of work-items in a single work-group through the use of a **work-group barrier**. All work-items within a work-group has to reach the barrier before any are allowed to continue execution beyond the barrier. OpenCL provides two forms of hierarchical data parallelism: data parallelism from work-items within a work-group, and data parallelism at the level of work-groups. Programmers are given control over defining the NDRange index space and the sizes of work-groups or leaving it to the system to decide.

Branch statements within a kernel can lead each work-item to execute very different operations. This is often known as the **Single Program Multiple Data** (SPMD) model. The kernel used in our LBP OpenCL program does not contain any branch statement; each work-item executes identical operations but on a subset of data items selected by its global ID. This case is known as the **Single Instruction Multiple Data** (SIMD) model.

The OpenCL execution model supports various task-parallel algorithms. The first version, also used in our LBP program, is when concurrency is internal to the task. A task is defined as a kernel that executes as a single work-item regardless of the NDRange used by other kernels. Many other versions include kernels submitted as tasks that execute at the same time with an out-of-order queue (by default, an in-order queue is always used); kernels connected into a task graph using OpenCLs event handling model.

because of the wide range of devices that OpenCL supports, there are parallel algorithm limitations to the OpenCL execution model. It is down to the reliability on assumptions made in the execution

model, such as when a kernel is executed, we can only assume that the work-items in a group will execute concurrently. Work-groups are free to run in any order including serially. This is also the case for multiple kernel executions. Even with an out-of-order queue (executes concurrently) enabled, an implementation is free to serialize the execution of kernels. Hence we cannot safely construct algorithms that relies on data being shared between work-groups servicing a single kernel execution. With multiple kernels, it is possible for early kernels to wait on events from later events, creating a deadlock.

3.3.5 Summary of OpenCL

The first step in OpenCL programming is the host application. This involves six different data structures: platforms, devices, contexts, programs, kernels, and command queues. Each of them provides two types of functions: one that creates it and the other that provides information about it after it has been created. The primary role of the host application is to send commands to devices. It usually starts by finding the chosen platform and its targeted devices.

The source code of kernel functions that provide the implementation of the program's algorithm must be build into a program before the host could dispatch it to one or more devices via the command queues. But to do their jobs, devices need more than just kernels, data is needed too. Thus overall three pieces of information are needed: the kernel instructions to be executed, a buffer containing the data to be processed, and another buffer where the output data will be stored. There are two types of memory buffer objects. Buffer objects store general data in a single dimension, and image objects store formatted pixel data in two or three dimensions. These data buffers are assigned to kernels by setting them up as arguments for the OpenCL kernel functions.

Next would be to get the most of the computational power of the target devices by telling them how to partition the data. Different devices may have different memory sizes and processing characteristics, therefore dividing data with respect to the targets' architecture would be the best. (Data partitioning will be explained in detail in the next section.) Finally the kernel is enqueued via the command queue to the device/s for processing.

3.4 Key OpenCL Concepts used in Implementations (of LBP in OpenCL)

Before we can get into the details of the LBP algorithm, recall from OpenCL's execution model that an application consists of a host program and a set of kernels for execution. The workings of the two stages (LBP image, LBP histogram) in the LBP algorithm will be in the kernel; but the processing cannot start without the base, that is, the host program set up. Data partitioning and image processing are some of OpenCLs key strengths in concurrency programming and crucial in our implementation. The SIMT architecture is the low level representation of what goes on with OpenCL and the GPU. A clear understanding of this will allowed us to optimize the most of our gains in this parallel computing approach. Local shared memory is key to maximizing the memory throughput from our GPU device; it is widely used a lot in many of today's GPU computing applications but our results in the evaluation section are most surprising.

3.4.1 Data Partitioning

One of OpenCL's most important capabilities, data partitioning, is crucial for any OpenCL application that demands high performance. The better the load is distributed, the sooner the computational tasks

will complete. The basic unit of work is the **work-item**, which corresponds to the code executed within a regular $C/C++$ loop.

```
for(a=0; a<4; a++) {
  for(b=2; b<8; b++) {
    for(c=3; c<12; c++) {
        process(point[a][b][c]);
    }
  }
}
```

In this loop example [30], the kernel would lie inside the innermost loop, that is, $process(point[a][b][c])$. A work item is considered as a single function call: $process(point[1][2][3])$.
Each work-item receives a **global ID** $\{a,b,c\}$ that allows it to access data specifically for it to process; so for work item $process(point[1][2][3])$, its global ID is $\{1,2,3\}$. The number of elements in a global ID is referred to as the **dimensionality** of the data. For the preceding example, it is 3. In our implementation, we will be dealing with a two-dimensional image as data thus its dimensionality would be 2. If work-items require synchronization in their data processing, they can be placed into **work-groups**. Each work-group executes on a single compute unit on the device where work-items are the processing elements of that compute unit. The most important capability of data partitioning is the ability to specify the sizes of the data that work-items will be processing on, and also the sizes of the work-groups. The sizes of the data are called **global work sizes**; this would be $\{4,6,9\}$ for our example above; again, the number of elements is according to the dimensionality of the data. So for a two dimensional image, we would need to specify two global work size values, for example one would be the width of the image and other, the height of the image. Or if we want only part of the image, we define the width and height of that region. The number of work-items in a work-group, that is, sizes of work-groups are set through the values of the **local work sizes**. In addition to the global ID, each work-item has a **local ID** that distinguishes it from all other work-items in the same work-group.

To take the fullest advantage of work-group partitioning, you need to understand how work-groups relate to device resources. The memory access between a processing element, that is, a work-item and its **private memory** is the fastest memory access on the device. **Local memory** access is slower, and **global memory** access is slower still. Therefore, for high-performance data processing, you need to use private and local memory as much as possible. When multiple work-items process the same data in local memory, their disordered execution can cause errors. To prevent these errors, we need to synchronize work-item execution therefore we use OpenCL **barriers** and **atomic operations**.
A barrier command prevents all following commands from executing until every preceding command has completed it execution. This forces a work-item to wait until every other work-item in the same local work-group reaches the barrier. For atomic operations, there are like any operation but they cannot be interrupted. It is highly useful when work-items access the same memory. As an example, work-item A and B both are executing $(x-=2)$ where x is set to 5. Work item A reads the value of x as 5. Work-item also reads the value of x as 5. Work-item A computes $(5-2=3)$ and stores 3 to memory. Work-item also computes $(5-2=3)$ and stores 3 to memory. The resulting value of x is 3 which is wrong; we wanted the answer to be -1. Therefore an appropriate atomic operation for this case would be $atomic\_sub(x,2)$. If work-items A and B were to execute $atomic\_sub(x,2)$, the

resulting x value would be -1. So atomic operations forces work-item B to wait until work-item A finishes, and successive work-items cannot start until after work-item B finishes.

3.4.2 SIMT Architecture and Hardware Multithreading

CUDA terminology will be used here; remember an OpenCL work-item is a CUDA thread and an Opencl work-group is a CUDA thread block. The SM multiprocessor creates, manges, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads of the same warp start together at the same program address, but they have their own instruction address counter and register state therefore are free to branch and execute independently [31].
When the SM is given thread blocks to execute, it partitions them into warps that are set up for execution by a warp scheduler. Each warp contains threads of consecutive, increasing thread Ids with the first warp containing thread 0. A warp executes one common instruction at a time, so there is full effiency when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, stopping threads that are not on that path, and when all threads have completed their paths, they converge back to the same execution path. Branch divergence can only occur within a warp; regardless of whether they are executing common or disjoint code paths, different warps execute independently.

If an atomic operation executed by a warp modifies the same location in global memory for more than one thread of a warp, every read and write to that location will be serialized, but the order in which they occur is random.

The number of blocks and warps that can reside and be processed together for a given kernel depends on the number of registers and amount of shared memory it needs, and the number of registers and amount of shared memory available on the SM. There are also a maximum number of resident blocks and resident warps per SM. These limits are a function of the device's compute capability; this will be further mentioned in the evaluation section where it is really important. If the available registers and shared memory is not enough for the SM to process at least one block, the kernel will fail to launch.

3.4.3 Image Processing

On GPUs, image data is stored in special global memory called texture memory. Unlike regular global memory, texture memory is cached for rapid access. When it comes to image processing in OpenCl, there are two primary data structures: image objects and samplers. Image objects serve as the storage mechanism that host applications use for pixel data transfers to and from a device. When the device receives the image data, samplers tells it how to read the image. GPUs makes use of the texture memory specifically for reading image objects. It is cached so that just one read is needed from the texture cache, and if there is a cache miss it would cost one memory read from the device memory. The texture cache is optimized for 2D spatial locality so that image addresses that are side-by-side can be read most efficiently by work-items of the same warp. It is also designed for streaming reads with a constant latency, that is, a cache hit reduces DRAM bandwidth demand, but not read latency. Many benefits comes from reading device memory through image objects, giving huge advantages over reading device memory from global or constant memory [17]. However, within the same kernel call, a work-item can only safely read via an image object some memory location if this memory location has been updated by a previous kernel call or memory copy. The image read will return undefined data if it has been previously updated by the same work-item or another work-item from the same kernel call.

Below in the kernel code of the implementations, you would see samplers are represented by *sampler_t* structure, and image objects are *image2d_t* structures for storing two dimensional image data. Our sampler properties used were *CLK_NORMALIZED_COORDS_FALSE* which specifies image coordinates are not normalized $(0.0 - 1.0)$, *CLK_ADDRESS_CLAMP* which set pixel values beyond the maximum dimensions of the two dimensional image to be black by default, and *CLK_FILTER_NEAREST* which specifies how pixel values are interpolated between pixels.

3.5 Implementation (of LBP in OpenCL)

There are two stages in the LBP algorithm: first, you get the LBP image by calculating for the LBP label of every pixel in the input image; second, would be to organize the labels appropriately to the histogram's respective 256 bins corresponding to the 256 different possible LBP labels. It is important to note that for performance considerations, the output of the LBP image is not necessarily if all we want is the histogram. The writing of LBP labels to a LBP image has some substantial costs in processing time and resources, which is not worth it since the production of the histogram requires only the label values.

3.5.1 LBP Image Generation

See Appendix A1 for the kernel code to produce the LBP image. Within it, *img_lbp* is the main function that calculates the LBP label for a given particular pixel at position (*ix, iy*) in the input image of width (*w*) and height (*h*). The input image is passed in as an OpenCL image buffer object (*img*) of type *image2d_t*. The output image buffer object is (*imgout*) of the same type *image2d_t*. Note that we do not need this output image buffer if we are not going to write to it as we mentioned before.

Firstly we get the given pixel's intensity value and that of its eight surrounding neighbours' too by using the locally-defined function *(read(image2d_t img, int w, int h, int2 coords)*. Within it we have applied a wrap-around concept for pixels that searched for outside of the input image; this occurs when we have center pixels that are on the borders of the input image, some of its neighbours would be out of range and therefore we wrap-around the image and use those pixels as replacements. *read_imageui*() is one of OpenCLs image processing functions, it is used to read the value of one pixel in the provided input image buffer object. sampler was mentioned in the Image Processing section before. Remember that the ordering of the neighbours is important, it has to be consistent.

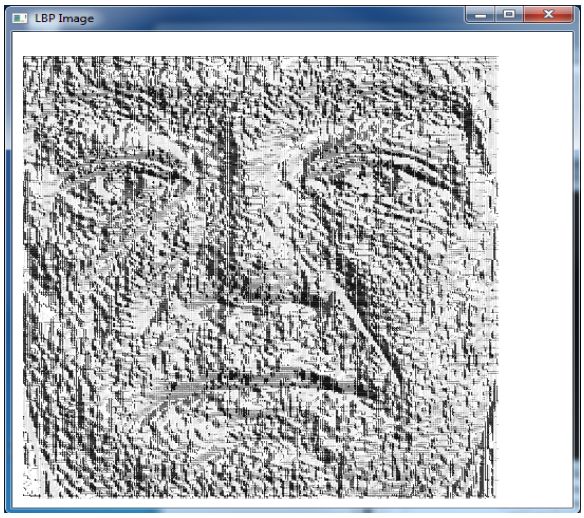
Next, after obtaining the nine pixels intensity values, we can perform thresholding on the eight pixels neighbours against the center pixel. The thresholded values either hold a value of 0 or 1. It is 0 if the respective neighbour pixel's intensity value is lower than the center pixel's, otherwise it is 1. The values are then multiplied with powers of two, the powers corresponds to the ordering of the neighbour. The first neighbour threshold value is multiplied with 2^0; the second neighbour's with 2^1 and so on till the eighth neighbour's with 2^7. These multiplied thresholded values are added together to produce the center pixel's LBP label.

If the LBP image output is desired, *write_imageui*(), one of OpenCLs image processing functions, can be used to write the LBP label to the output image buffer object (*imgout*). A huge part in this kernel function's performance is relied on OpenCL image processing capability with the GPU device's accesses to texture memory. There are not many other options that can be taken to give a major

improvement in execution. But still it worked well. The correctness of the program was tested with Figure 3.6(a) of an input image example and the resulted LBP image Figure 3.6(b) output from it; they were taken from [15]. Figure 3.6(c) is our implementation's LBP image output when Figure 3.6(a) was used as the input image data. You can see that it is exactly like Figure 3.6(b), therefore we can be assured that the LBP image generating implementation is correct.

The *img_lbp* kernel function is called from the main kernel function which handles the histogram output described in the next section. In that case we would comment out the *write_imageui*() function as we only need the LBP histogram and not the image output. The input image is data partitioned and global work sizes chosen such that one work item only has to calculate the LBP label of one pixel in the image. The workings of data partitioning, work-group sizes and IDs will be clearer in the next section where they are crucial for generating the histogram.

(a) Input Face Image example from [15]



(b) Output LBP Face Image example from [15]

(c) Our Implementation's LBP Face Image Output

Figure 3.6: Face images

3.5.2 LBP Histogram Generation

There are various ways this can be done, with possible alterations made to the kernel and host program giving different advantages and disadvantages. Data partitioning with the different local and global work sizes can give deviations in the time spent for processing; getting the right sizes can be tricky but easier with the right tools at hand. There are three methods that were implemented here: 1)By only accessing OpenCL global memory 2)By accessing OpenCL global and local memory 3)MapReduce.

To set-up, we need the host application to transfer an empty buffer object to the device; this buffer object will be containing the LBP labels to form the LBP histogram. We allocate this buffer object to be an array of size 256 corresponding to the 256 bins of the LBP histogram; and also this buffer object memory is stored in the device's global/constant memory after the transfer. The global/constant memory is commonly the largest region in the device's memory but it also has the lowest memory bandwidth.

It is known that work-items access local memory much faster than they can access global or constant memory. Local memory is not nearly as large as global or constant memory, but because of its access speed, it is a good place for work-items to store intermediate results of a kernel execution. However different work-groups have different regions of local memory so only work-items of the same work group can access the same block of local memory. Also a host application is not able to write or read to a device's local memory space. Therefore it is recommended that work items read their dedicated part of the global memory into local memory and collaboratively with other work items in the same work group, process the data there. Having multiple work-items processing the same data helps improve performance. This would definitely require work-item synchronization so as in many concurrent applications. After finished processing the local data, they can write their results back to global memory which can be transferred back to the host. Thus the first two implementations, to test the gains and losses from using global memory alone or with local memory. MapReduce is different level of application, it has a framework for solving classes of problems using distributed processing. It involves more than just considering the number of global and local memory accesses.

For the production of the LBP image shown in the previous section 3.5.1, all of the variables are defined within a work-item's private memory except for the image buffer objects which are in the device's texture cache memory. A work-item can access this memory faster than it can access local memory or global/constant memory.

3.5.2.1 Only OpenCL Global Memory Accesses

A work item is an execution of the main kernel function (*__kernel void histogram_image()*), that is, a single implementation of the kernel on a specific set of data. The (x and y) values are the work-item's global ID which uniquely identifies it among other work-items; and allows it to access its specific set of data for processing. In the host program, we have partitioned the two dimensional input image such that its total number of pixels is equal to the number of work-items. Normally when these work-items are generated for a kernel, they execute in a disordered, non-deterministic fashion. This is fine since work-items access different texture memory regions of the image. And we rely on the atomic increment operation ($atomic\_inc()$) when work items write the LBP label of their pixel into the 256 bins histogram buffer object (*__global uint* histogram*).

See Appendix A2 for the kernel code in this implementation. Notice that the $img\_lbp()$ function we defined in the previous section is used here within the if condition loop, to get the LBP label of the pixel.

3.5.2.2 OpenCL Global and Local Memory Accesses

There is a huge bit of more code in Appendix A3 to utilise the shared local memories of the many work-groups on the GPU device. Basically just like how we need the global ID (int x, int y) of the executing work-item to differentiate it from other work-items, we also need its local ID (int tid) to individualize it among its local work-items within the same work-group; this is also so that we could coordinate how the workings using the shared local memory would be done. The first do-while condition loop is to clear the local temporary histogram ($\_\_local\ uint\ tmp\_histogram[256]$) such that its values are all zeros. It looks complicated because work-items all work in parallel therefore we are utilising their local ID to parallely clear the local histogram.

Next, to synchronize the work-items in local memory access, we use a barrier function with the appropriate flag ($barrier(CLK\_LOCAL\_MEM\_FENCE)$). The first barrier function forces every work-item in the same work-group to finish clearing their local histogram before starting to calculate for the LBP labels. The second barrier is to ensure every work-item has got their LBP label and incremented the appropriate bin of the local histogram, before the data transfer of the local histogram's 256 bins to global histogram can start. Again, it looks complicated because for concurrency purposes, we utilise the work items' local IDs to do the transferring. Notice the atomic addition operation ($atomic\_add()$) used to do the carry the 256 LBP bin data to the global histogram. Both the local and global histogram is of size 256.

3.5.2.3 MapReduce

MapReduce is purely a theoretical framework for building distributed applications that process huge amounts of data. Google relies on it for its high-speed internet data analysis [30]. MapReduce does not focus on any particular processing task, but presents an approach that can be applied to multiple processors. It contains a minimum of two stages: mapping and reduction. The processors involved in the mapping stage each receive input data and produce key-value pairs. Once the mapping is finished, each processor in the reduction stage receives values corresponding to a given key. The processors performing reduction process these values and merge them together to form the output data. This processing independence is key to MapReduce's efficiency because it means the processors do not have to wait for other tasks to complete.

In relation to OpenCL, the processors are just the work-items. And as we have seen a lot previously, work items access local memory faster than global or constant memory therefore the mapping and reducing operations will mostly be all done locally. A MapReduce data set is much too large to be processed by a single work-group with its local memory, therefore multiple work-groups will be used instead with their results combined in global memory. In a key and value pair, the key would range from 1 to 256 corresponding to the 256 bins in the local LBP histogram and the value would be the level of that bin. Figure 3.7 shows one method of implementing mapping and reduction on an OpenCL-compliant GPU device. It depicts two work-groups, each with three or more work-items. The MapReduce implementation method presented in the figure splits the reduction stage into two sub-stages: local reduction and global reduction. Local reduction, like the mapping stage, operates

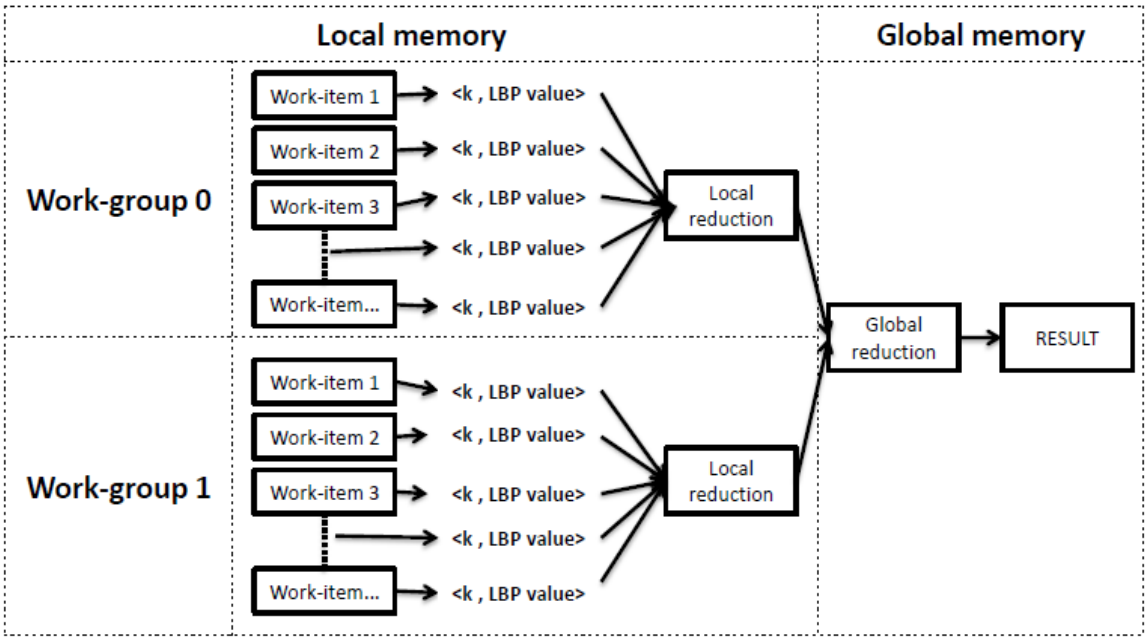


Figure 3.7: MapReduce Processing Model in OpenCL

on the data in local memory space and saves the results to global memory which global reduction receives before producing a result to be sent back to the host program.

Everything in the kernel code in Appendix A4.1 is mostly the same as the implementation using global and local memory accesses. The only difference is after the second barrier function where the operations are still very similar but we no longer use the atomic addition, instead we have a much bigger sized global histogram. The size of the global histogram would be 256 times the number of work-groups, that is, for every work-group, there are 256 slots for their local work-items to increment with respect to their calculated LBP label value. Thus when you see the transfer of data to the global histogram, its index (*histogram[group_indx + indx + tid]*) has the *group_indx* value in it. Each work-group has their own unique *group_indx* value. You could see this global histogram to be more like a global memory container with all LBP label data organized properly according to work-group index. The local histogram is still the same, 256 bins long.

This is not all, we still need to combined this long array of LBP label data into a 256 bins histogram; this is done by the next kernel in Appendix A4.2 . We have two kernels for MapReduce, unlike the first two implementations having only one kernel. This second kernel's function name implies what it does, it sums up the LBP labels within the global partial histogram (*global uint *partial_histogram*) and saves it to the 256 bins global histogram (*global uint *histogram*). The global partial histogram is the same global histogram from the previous kernel, it just has a different variable name here. There are definitely costs such as execution overheads when processing two kernels instead of one. But it was suggested as a technique for image histogram generation in a OpenCL programming guide [32].

4 Evaluation

4.1 Graphics Processing Units (GPUs) used

Two GPUs by NVIDIA were used as our experimenting devices to run the OpenCL implementations on: NVIDIA GeForce GTX 650, a desktop GPU and NVIDIA GT 740M, a laptop GPU.

MEMORY	**CLOCK SPEEDS**	**RENDER CONFIG**
Memory Size: 1024MB	**GPU Clock:** 1058MHz	**Shading Units: 384**
Memory Type: GDDR5	**Memory Clock:** 1250MHz (5000 MHz effective)	**SMX Count: 2**
Memory Bus: 128 bit		**Pixel Rate: 8.46 GPixel/s**
Memory Bandwidth: 80.0GB/s		**Texture Rate: 33.9 GTexel/s**
		Floating-point performance: 812.5 GFLOPS

Table 4.1: Device specifications of NVIDIA GTX 650 [33]

=

MEMORY	**CLOCK SPEEDS**	**RENDER CONFIG**
Memory Size: 2048MB	**GPU Clock:** 980MHz	**Shading Units: 384**
Memory Type: DDR3	**Memory Clock:** 900MHz (1800 MHz effective)	**SMX Count: 2**
Memory Bus: 64 bit		**Pixel Rate: 7.84 GPixel/s**
Memory Bandwidth: 14.4GB/s		**Texture Rate: 15.7 GTexel/s**
		Floating-point performance: 752.6 GFLOPS

Table 4.2: Device specifications of NVIDIA GT 740M [34]

SMX units are the new names of enhanced SM units in the latest Kepler GPU micro-architecture. The difference between SM and SMX units is significant. NVIDIA reduced the amount of control logic and increased the number of CUDA cores 6-fold to 192. This increases the efficiency (performance per watt) of the Kepler SMX unit to twice of that of the SM unit from its preceding Fermi GPU micro-architecture.

4.2 Measurement tools

Profiling is a vital tool in high-performance application development as it allows us to evaluate the performance of computing hardware and coding methods. With profiling, we could compare our two GPU devices, the different implemented kernels, and data partitioning strategies. There are plenty of performance analysis tools, developed platforms and APIs that can be used to profile running applications:

1. QueryPerformanceCounter (QPC)
2. OpenCL Profiling
3. NVIDIA Compute Visual Profiler
4. NVIDIA Nsight

QPC and OpenCL profiling functions are closer to the run and bolts of the application, using reliable and efficient performance counters to provide time-stamps and time-interval measurements. The NVIDIA visual profiler is a standalone cross-platform application allowing programmers to visualize an application's activity on a time-line so that opportunities for performance enhancements can be easily found. For NVIDIA Nsight, the Microsoft Visual Studio edition was used; it is similar to the NVIDIA visual profiler just that it is integrated into the Visual Studio IDE. Initial experiments were ran with QPC and OpenCL profiling, producing results that required explanations. This was hard with the limited information on the application's executions and interactivity with the CPU and GPU. Hence more powerful analysis instruments such as NVIDIA Visual profiler are needed. Both were also used to double confirm time measurements obtained from QPC and OpenCL profiling too.

4.2.1 Query Performance Counter (QPC)

QPC was used to profile only the host program. However not everything in the host program were being kept tracked of, only from the start of the creation of the image buffer objects and histogram buffer memory objects, to the reading of the output histogram memory buffer. Note that the set up of the platforms and devices are not included so as the reading of the output LBP image as the LBP histogram is the only desired output.

QPC is the primary API for windows used to acquire high-resolution time stamps or measure time intervals. It is independent of and is not synchronized to any external time reference. These time measurements involves the computation of response time, throughput, and latency as well as profiling code execution; and they are defined with a start and end event in any section of the program code. Its performance counter is accurate to micro-seconds.

QPC could have been also used to profile kernel executions but it is rather obvious that OpenCL's inbuilt profiling would perform a more accurate reading since the program's code is developed in the same OpenCL environment.

4.2.2 OpenCL Profiling

We have seen that data partitioning allows the division of kernel execution into multiple work-items. Therefore measuring the kernel execution times would help rate the effectiveness of the data partitioning strategy. OpenCL profiling was done only on kernel executions and not whole program executions

as only events that are sent for processing via the command queue can be profiled. Therefore events such as initialising the right variables and setting up the image buffer objects were not possible to take time measurements of.

To obtain timing information about the execution of a kernel, we profile its en-queueing into a pre-defined command queue till it returns from execution [30]. This is done by first creating a command queue with profiling enabled, allowing OpenCL to record whenever commands in the queue change state. Next we associate an event with the kernel enqueue command; this event will store the timing data for this command. Lastly after the kernel enqueue command completes its execution, we call an OpenCL profiling function to access the event and obtain timing information of its associated kernel command. The profiling times are roughly given in nano-seconds, but of course whether this is done to individual seconds depends on the device's resolution.

4.2.3 NVIDIA Compute Visual Profiler and NVIDIA Nsight

The NVIDIA Compute Visual Profiler 4.0 was developed to monitor CUDA applications, but with a few tweaks OpenCL applications can be analysed too. NVIDIA's CUDA tool-kit 4.0 does provide support to profile applications built in the OpenCL framework. The visual profiler is available as both a standalone application and as part of Nsight Visual Studio Edition. The standalone version exists within the NVIDIA CUDA tool-kit. Both offers powering debugging and profiling tools such as guided analysis results, and an application time-line showing CPU and GPU activity with specific metric and event values revealing information like how the kernels in an application are behaving.

4.3 Implementation Aspects

In our implementations, the total number of work-items was the same as the number of pixels of the input two dimensional image data as this was trivially the optimal number, however it was not clear what size the work-groups of these work-items should they be. A work-group is a combination of work-items that access the same processing resources. Work-items in a work-group have high-speed access to the same block of local memory, allowing for execution synchronization to give better performance efficiency.

Each work-group executes on a single compute unit, and each compute unit can only execute one work-group at a time. Therefore regardless of how many work-items and work-groups are generated, if the GPU device only has M compute units and N processing elements per compute unit, then only MN work-items will execute the kernel at any given time. Then again, to make the best use of a work-group's local memory, it is important to have as many work-items in a work-group as possible. Therefore one of the aims in these experiments is to find the optimal work-group size with relation to the GPU device's specifications.

In theory, work-items access local memory much faster than global/constant memory therefore a large number of global memory accesses and a small number of local memory accesses should take longer than a small number of global memory accesses and a large number of local memory accesses. This would mean that the implementation using only global memory accesses should be the slowest.

For the MapReduce approach, it was unclear how much the overheads of transferring data between kernels would affect the total execution time. And whether using a huge partial histogram memory

buffer to store LBP label data of work-groups into their specific work-group locations is a benefit though it does not require to use any atomic operations. These atomic operations can only be processed by one work-item at a time, other work-items have to wait for their turn. This implied that the implementation using local and global memory accesses might be slower than MapReduce because of its dependencies on atomic operations. Recall that MapReduce could use regular operations instead of atomic operations because of how its partial histogram memory buffer in the first kernel, was designed to hold the LBP label histogram data. Only by fully investigating each implementation's trade-off , can we then find which one would be fastest.

4.4 Experiment and Results

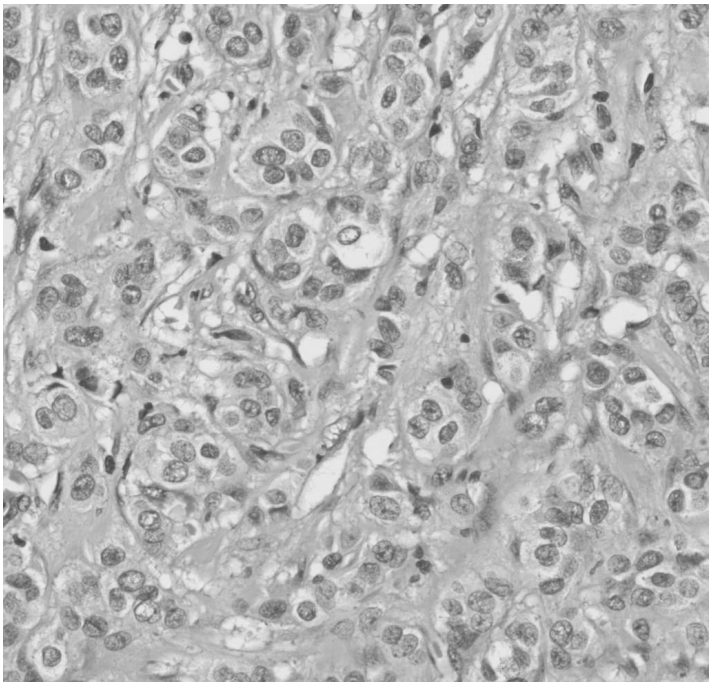


Figure 4.1: High resolution biopsy tissue image

There were two stages of experiments carried out; initial comparisons were made between a serialized LBP implementation running on a CPU and several concurrent LBP kernels executed on the NVIDIA GT-740M device, to measure and confirm the expected performance gains from GPGPU computing. Multiple tests were ran with various independent variables such as work-group sizes and sizes of input image to provide an all-rounded comparison between the different implementations running on the GPU. As was mentioned before, NVIDIA's performance profiling tools such as NVIDIA Compute Visual Profiler and other NVIDIA GPU computing tools, were utilised to provide logical explanations to results produced from this first stage of tests. This also prompted further research into optimizing the GPU device, considering not only the algorithmic side of the implementations but also the fixed number of resources that are available on the device and how best to make use of them. This motivated for a performance-to-device's-resources comparison thus the second stage of experiments was to study two different GPU devices and their executions. This was also a chance to see OpenCL's heterogeneous parallel programming portability in action though the compared device was still a GPU and made from the same manufacturer.

Large resolution images are one of the many goals in this research, therefore images of biopsy tissue sample ranging in sizes 1024x1024 to 8336x8336 were used as inputs to the programs. All programs were targeted to output only the overall LBP histogram, therefore initializations, memory readings and writings associated with generating the LBP image were taken out. But with LBP image generation enabled, we get Figure 4.1 and 4.2 which are the input and LBP output images respectively.

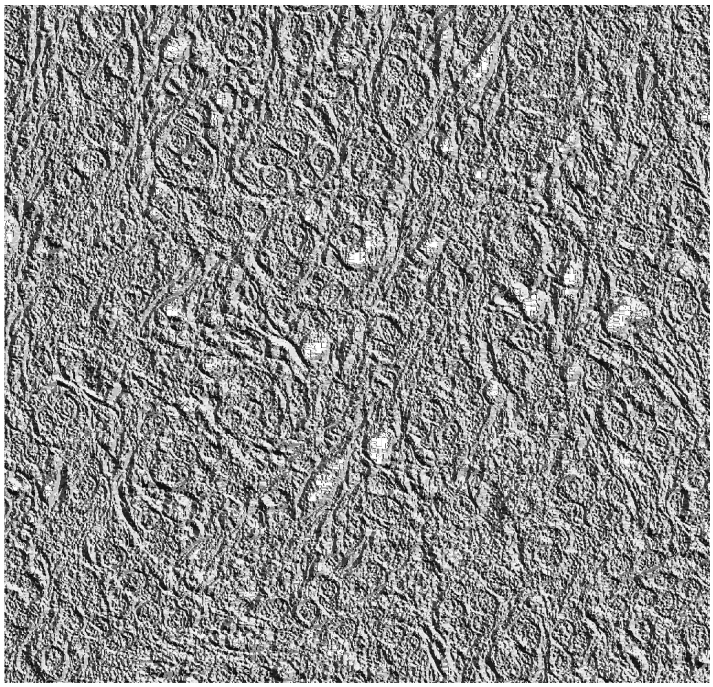


Figure 4.2: LBP output of high resolution biopsy tissue image

4.4.1 CPU specifications

The laptop CPU tested with NVIDIA GeForce GT740M as its GPU, has an Intel Core i5-4200M CPU @ 2.50GHz, with 4GB RAM running on a Windows 8.1 64-bit operating system.
NVIDIA GeForce GTX650 GPU is on the same machine as a desktop CPU with specifications: Intel Core i7-3770 CPU @ 3.40GHz, 8GB RAM, and runs on a Windows 7 64-bit operating system.

4.4.2 CPU and NIVIDA GT740M GPU

It is fair to say anyone would expect a program executed serially would take an exponential growth in computing time with the increase in image size, but processing times for concurrent programs would only rise slightly. Overall results presented the parallelized implementations to be faster than the serialized one. The growths in execution times were not surprising but there were a few noticeable measurements in Figure 4.3; firstly the CPU had a shorter program execution time than the MapReduce approach when the image size was 1024x1024; secondly the opposite of expected results happened, the program using only global memory accesses was found to be the fastest out of all three parallelized implementations; using a combination of local and global memory accesses was next fastest and MapReduce took the longest to process. Figure 4.4 shows a closer-up view of the three implementations. A fixed work-group size of 128 was used.

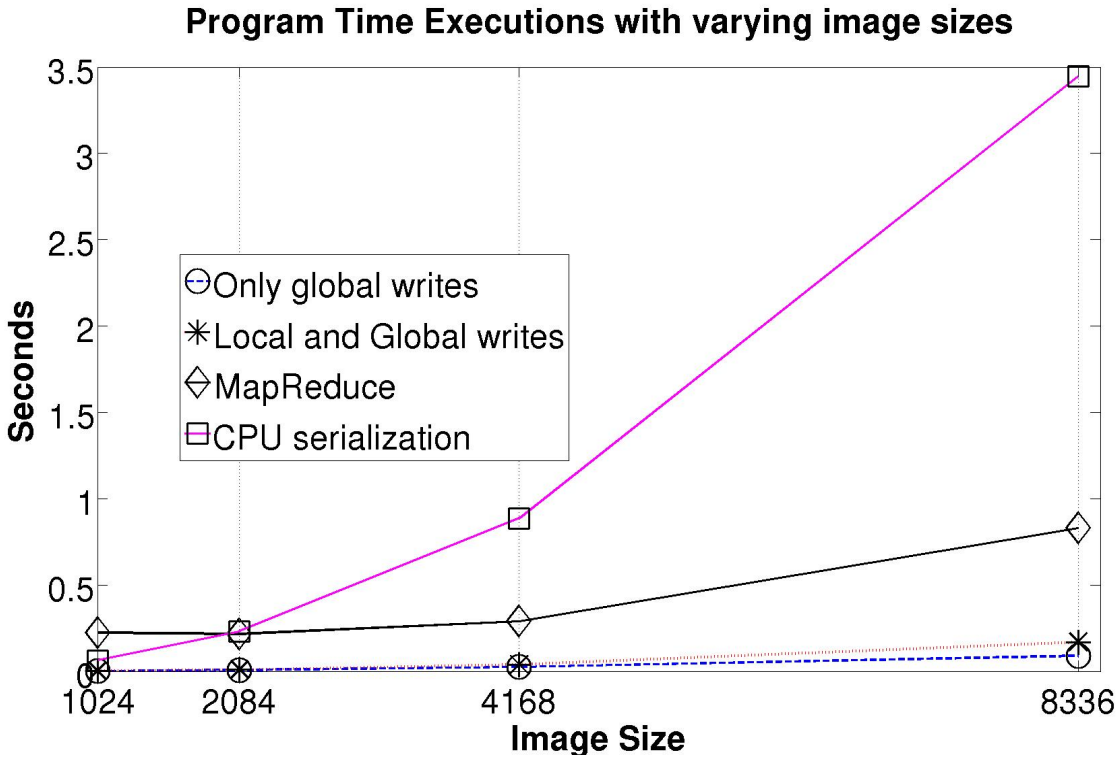


Figure 4.3: Program Time Executions - CPU and GT740M

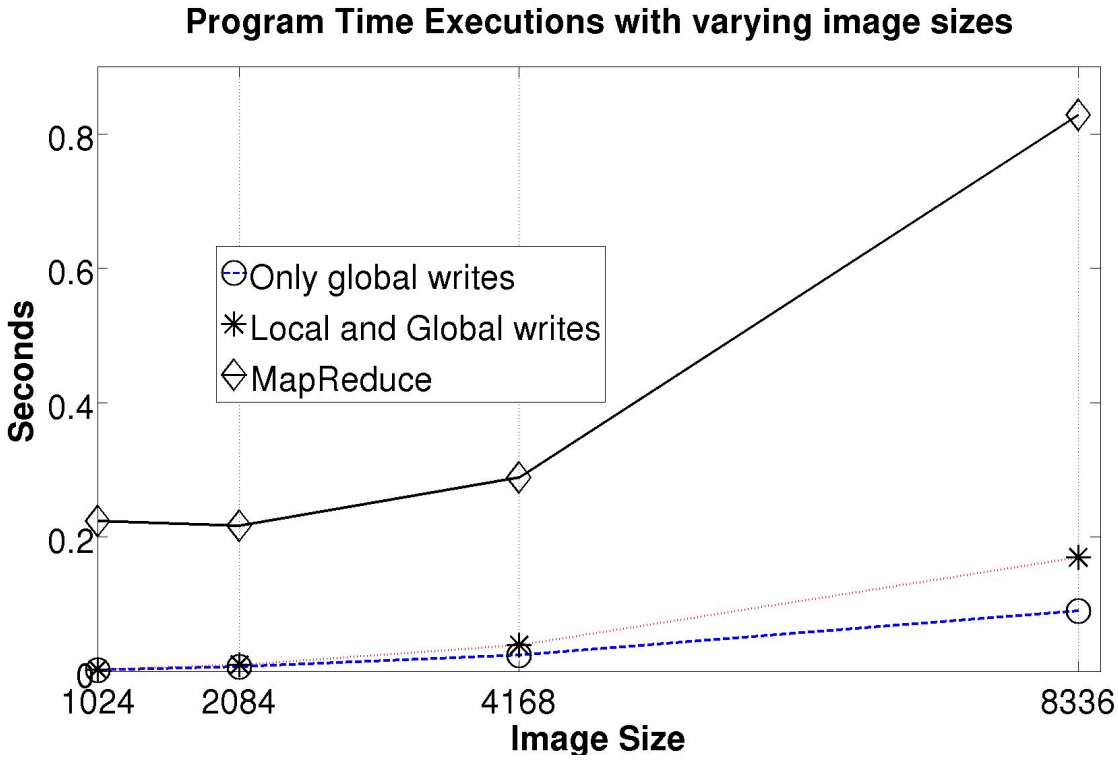


Figure 4.4: Program Time Executions - GT740M

To understand why the plots are like they are, more following tests were done, such as looking into the kernel execution times with varying work-group sizes, kernel execution times of a kernel using solely local memory accesses and another using only global memory accesses, and program execution times of the different implementations. All these for attempting to narrow down the operations' processing within the kernels and host program, to find the main factor affecting time performance.

4.4.3 Implementations on GT740M GPU

For tests without varying image size as their independent variable, a fixed image of size 8336x8336 was used as input image data. From Figure 4.5 we can conclude that too many work-items in a work-group was not going to help much even in implementations utilising huge local data transfers and processing. Furthermore we saw that global memory access appeared to be taking the shortest execution times regardless of the work-group size. And despite the limited number of compute units on the GPU device, the overheads of having many small processing work-groups waiting for their turn to execute was nothing compared to the slow local memory accesses in a few large work-groups.

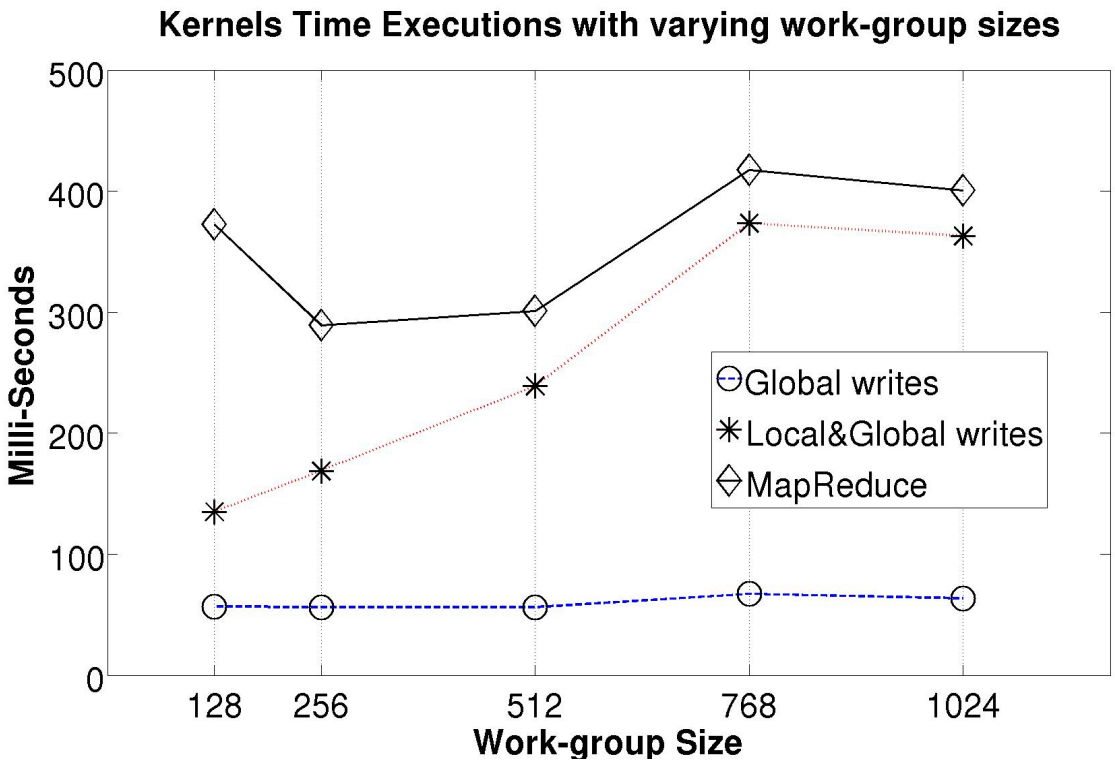


Figure 4.5: Kernel Time Executions with varying work-group sizes - GT740M

There is also some kind of trend among the three implementation plots: as the work-group size increases, the kernel execution times increases however after work-group size of 768, the time decreases. Aside from the abnormality at work-group size of 128 in MapReduce, this trend is apparent at every point. MapReduce relied heavily on local memory transfers and processing therefore having more work-items per work-group makes the workload lighter. But like was mentioned earlier, there were trade-offs in having a large work-group which started to show significantly from work-group size 512

onwards. The ideal work-group size for MapReduce can be deduced from this plot to be 256, while the other two clearly worked really well with just 128 work-items in a work-group. The reason behind the time differences between MapReduce and the implementation with local and global writes despite their common dependencies on local memory accesses was the possible overheads in data transfer between the device and host, and the executions of two kernels in MapReduce.

The memory accesses in the plots are called writes, as the programs executed have to write LBP label increments into either a locally defined histogram or a globally defined histogram; this is dependent on the type of implementation.

There was a vast difference between the implementation solely using global writes to that using both local and global memory accesses. To truly see why that was so meant looking closer at the time differences between local and global memory accesses, thus we have the next plot. To produce Figure

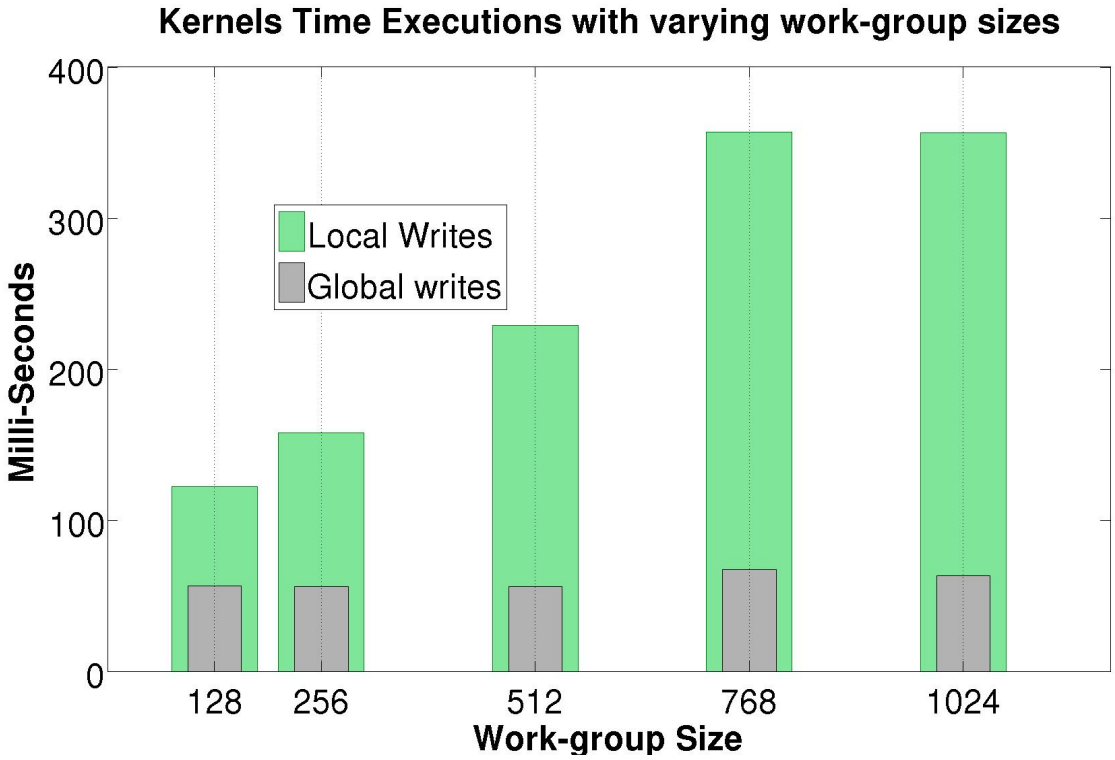


Figure 4.6: Kernel Time Executions of GT740M's global and local memory accesses

4.6, we modified the first implementation to produce two separate programs with one writing LBP label increments into a locally defined histogram and the other into a globally defined histogram.

From the plot, it is obvious that global memory accesses takes much lesser time to be processed compared to local accesses. However this contradicts the theory previously mentioned in the OpenCL sections that work-items access their shared work-group local memory much faster than they can access global/constant memory. Further tests were done and if you could recall in Appendix A2 and A3, the atomic increment function $(atomic\_inc())$ was used to perform the LBP histogram label increments $atomic\_inc(\&histogram[clr])$, $atomic\_inc(\&tmp\_histogram[clr])$. This was the root cause of

the slow execution time in local memory space; tests did show that shared memory writes does avoid uncoalesced memory accesses, leading to faster execution times. However this was done having work items access different regions of memory like the MapReduce approach. Problems can arise if multiple work items access the same memory at the same time therefore the need to use atomic operations. Without atomic operations, normal operations like $(histogram[clr]++)$ or $(tmp\_histogram[clr]++)$ have to be used instead, leading to inaccurate results and producing a wrong LBP histogram as output.

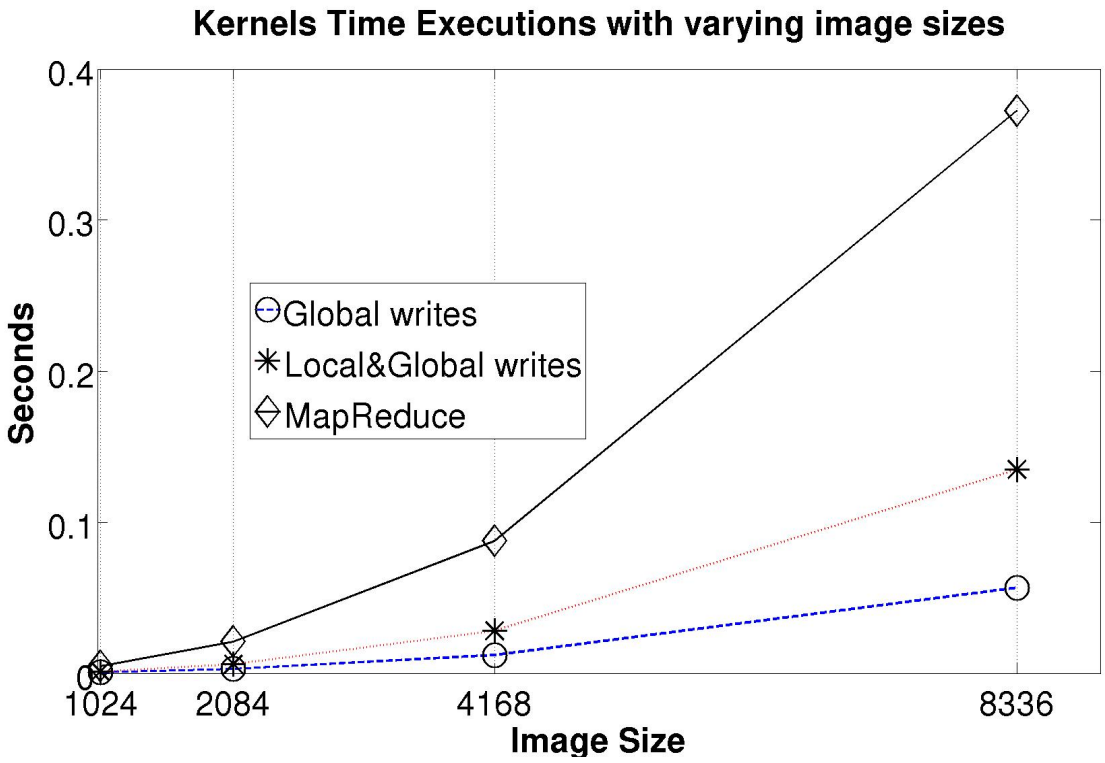


Figure 4.7: Kernel Time Executions with varying image sizes - GT740M

Nothing surprising is seen in Figure 4.7, it is expected for time executions to be longer with a huger input image data. The growth rates are believable too. Compared to the other two implementations, MapReduce had to carry huge costs with large data transfers to and fro between the device and host program; and it had to do it twice for each kernel, that is four in total for its two kernels. It is true that MapReduce's global partial histogram was large with different memory regions for different work-groups, meaning normal operations can be used to transfer the local histogram labels data to this global memory buffer. However MapReduce relied heavily on work-items in the same work-group to do the local processing, that is, incrementing their local histogram in the shared work-group memory which normal operations cannot be used but atomic increment operations. Therefore MapReduce had massive increases in execution times as the input images grew larger.

4.4.4 Compute Capability and Optimal Work-Group Size

A SM's registers and shared local memory are split among all the work-items of the active processing work-groups. The number of active blocks per SM, also refers to the number of work-groups a SM

can process at once, is dependent on how many registers per work-item and how much shared local memory per work-group are required for a given kernel.

NVIDIA GT740M's compute capability is version 3.5 [34]. The compute capability of a device is represented by a version number, also sometimes called its "SM version". This version number identifies the supported features by the GPU hardware and is used by applications at run-time to determine which hardware features and/or instructions are available on the GPU. The compute capability version comprises a major and a minor version number. GT740M is of Kepler architecture and so does GTX650 therefore they both have a major version of 3. GTX650's compute capability is 3.0

This compute capability version affects largely on the performance, execution and memory models of the device. This is when one of NVIDIA's GPU computing tool-kit utilities becomes most useful, the CUDA occupancy calculator. Using this calculator, one can see the physical limits of the GPU and make changes to the dependent resource usages (such as threads per block, registers per thread, shared memory per block) to see the calculator's derived expectancy of the device's performance.

Physical Limits for GPU Compute Capability:	**3.0**	**3.5**
Threads per Warp	32	32
Warps per Multiprocessor	64	64
Threads per Multiprocessor	2048	2048
Thread Blocks per Multiprocessor	16	16
Total # of 32-bit registers per Multiprocessor	65536	65536
Register Allocation unit size	256	256
Register Allocation granularity	warp	warp
Registers per Thread	63	255
Shared Memory per Multiprocessor(bytes)	49152	49152
Shared Memory Allocation unit size	256	256
Warp Allocation granularity	4	4
Maximum Thread block size	1024	1024

Table 4.3: CUDA Occupancy Calculator [35]

To maximise the GPU, we have to optimize the kernel launch configuration for the best possible occupancy in the GPU, that is, filling up the occupancy of each SM on the GPU device. Both GT740M and GTX650 has two SMX each so to maximise either GPU, we require to have as many active work-items, active warps and active work-groups per SMX. Using the NVIDIA Compute Visual Profiler, it was found that the first LBP implementation kernel was using 19 registers per work-item. Plugging this value into the resources usages of CUDA occupancy calculator, we found that having 128 work-items per work-group maximised all physical benefits of the SMX. A work-group size of 128, 256, 512 and 1024 work-items all occupied the SMX fully 100% however only size 128 used all of the present 16 compute units in the SMX while size 256 used only 8, 512 used only 4 and 1024 used only 2. Therefore having 128 threads per block in NVIDIA GT740M of compute capability 3.5 was its optimal work-group size for this kernel.

Remember the trend in Figure 4.5, that drop in execution time from work-group size 768 to 1024 in all three parallelized implementations, this was due to the occupancy of the SMX. Having 768

work-items per work-group actually only occupied 75% of the SMX, therefore not producing the full performance capability of the GPU and therefore took a longer time to execute.

The only physical limit difference for the GPU between compute capability 3.0 and 3.5, is the number of available registers per work-item. Refer to Table 4.3 . In version 3.5, there are 255 registers and 63 registers in version 3.0 . Thus the performance of GTX650 was expected to not be that different compared to GT750M, at least from the GPU occupancy point of view.

4.4.5 A Comparative Analysis between GT740M and GTX650

Other than having a different GPU compute capability, GTX650 has a faster memory clock, GPU clock, texture and pixel rate too. These distinctions may or may not give a significant difference in performance; to view this we have the next following plots in kernel and program execution times. Again, if varying image sizes is not the dependent variable of a plot, an image size of 8336x8336 was used as input for that test; and if varying work-group sizes is not the dependent variable, a work-group size of 128 was used because size 128 was found to be the optimal number for GPU GT740M.

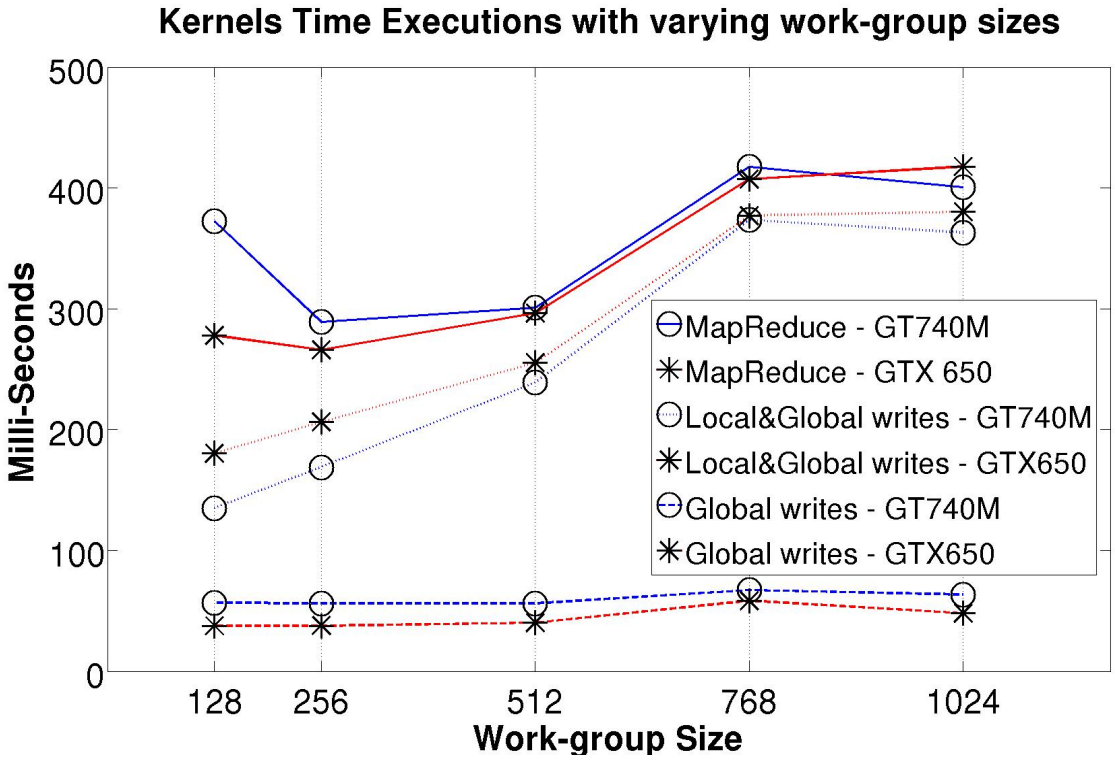


Figure 4.8: Kernel Time Executions with varying work-group sizes

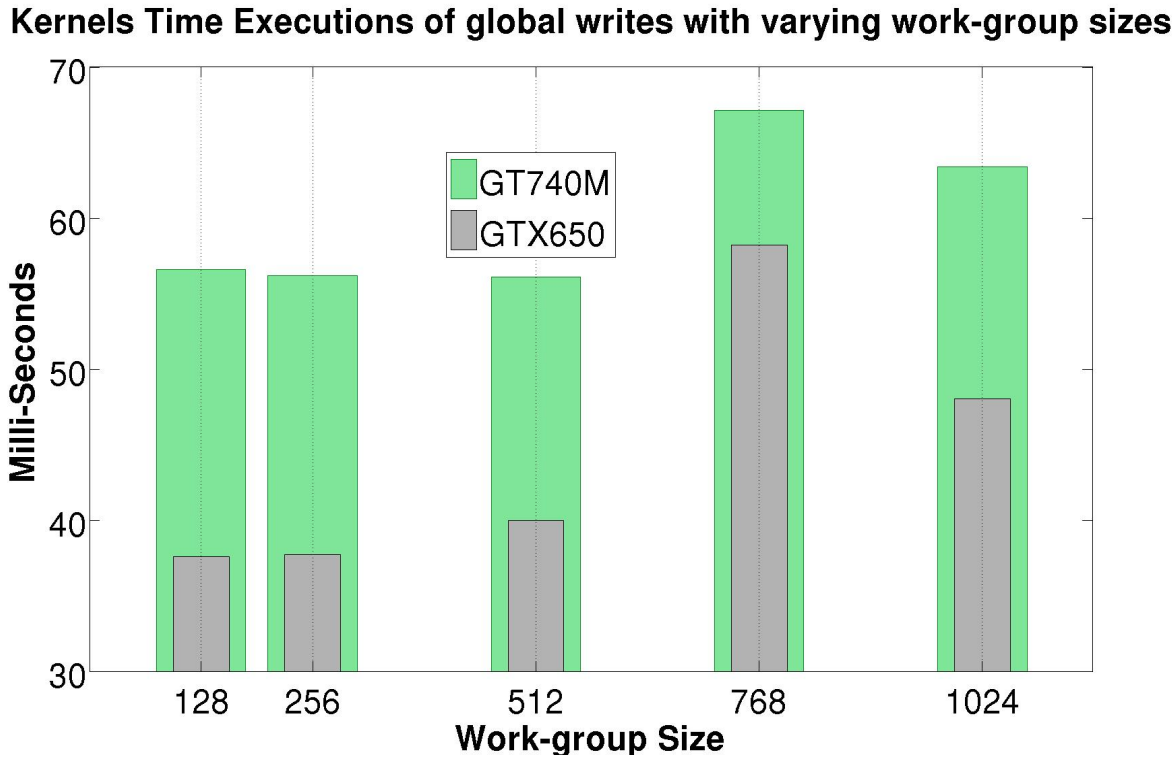


Figure 4.9: Kernel Time Executions with global memory accesses

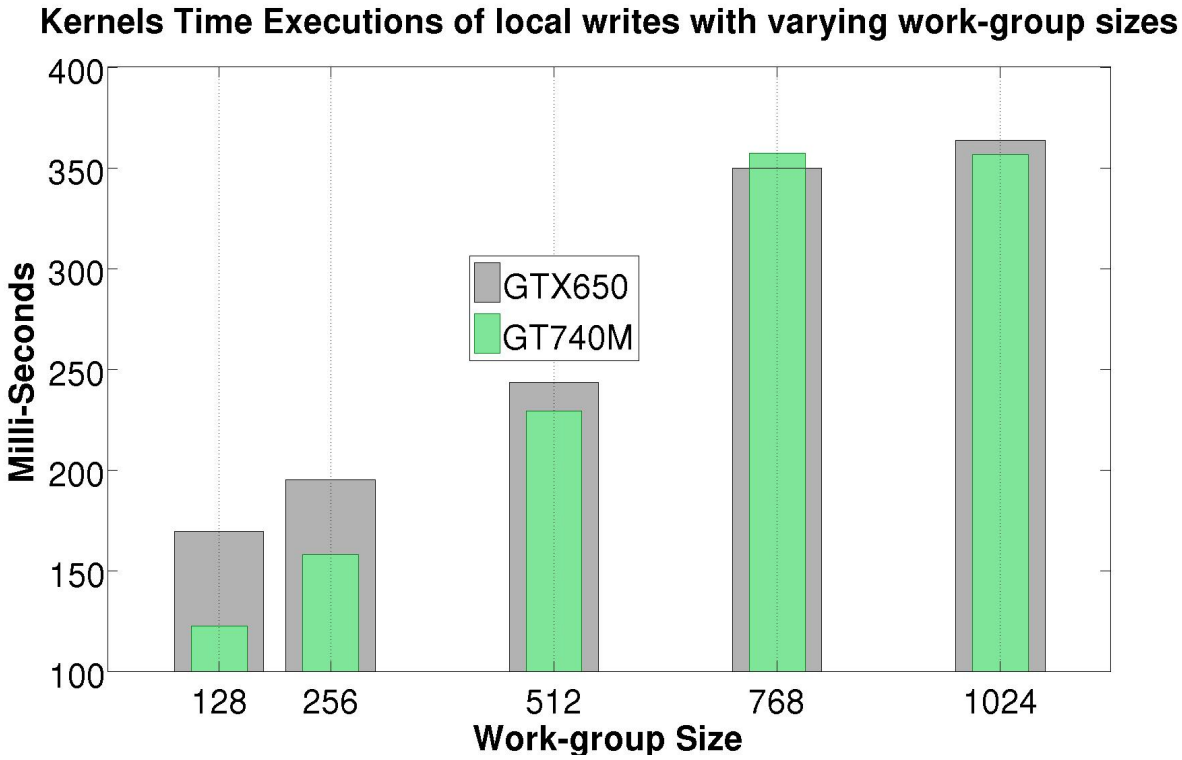


Figure 4.10: Kernel Time Executions with local memory accesses

Figure 4.9 and 4.10 certainly provides some interesting insights into the performance behaviour shown in Figure 4.8 . GT740M seemed to be executing the kernel using local memory accesses faster than GTX650 while GTX650 was quicker processing global memory accesses. This explained why the implementation with global memory accesses only was processed faster on the GTX650, and also partly why the implementation with both global and local memory writes was quickest on the GT740M device. MapReduce was similar to the latter implementation in the type of memory accesses however its operations relied on the transfers of data from the host program to the device and vice-versa. GTX650 has a much faster memory clock speed and bandwidth therefore this might be why it processed the MapReduce approach slightly quicker than GT740M.

The only device resource comparisons that were done are in Table 4.1 and 4.2 . NVIDIA Visual Profiler could have been used more efficiently to provide a deeper analysis but there was not enough time left to do so. Therefore not much can be explained about Figure 4.9 and 4.10 .

Figure 4.11 and 4.12 shows both the kernel and program time performance readings with varying input image sizes respectively. The behaviour shown in the comparisons of the different implementations between the two GPU devices is similar to Figure 4.8; with GTX650 faster in processing MapReduce and the first implementation using global memory accesses while GT740M performed better with the second implementation that operated with both local and global memory spaces.

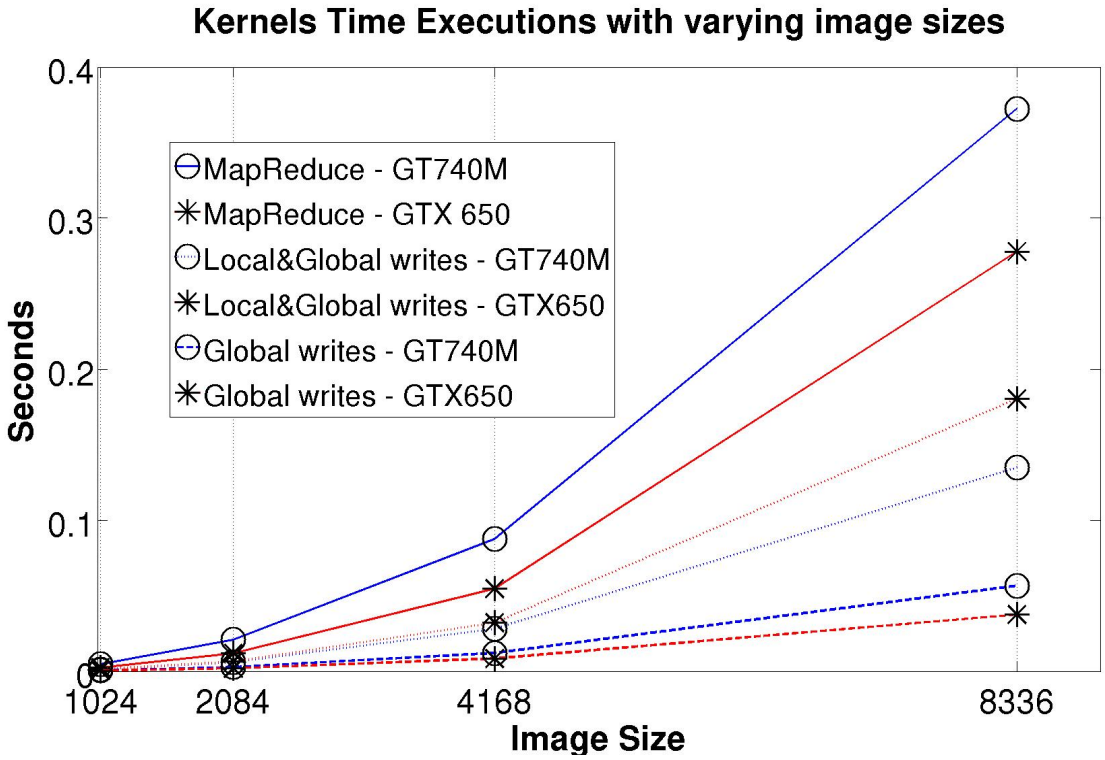


Figure 4.11: Kernel Time Executions with varying image sizes

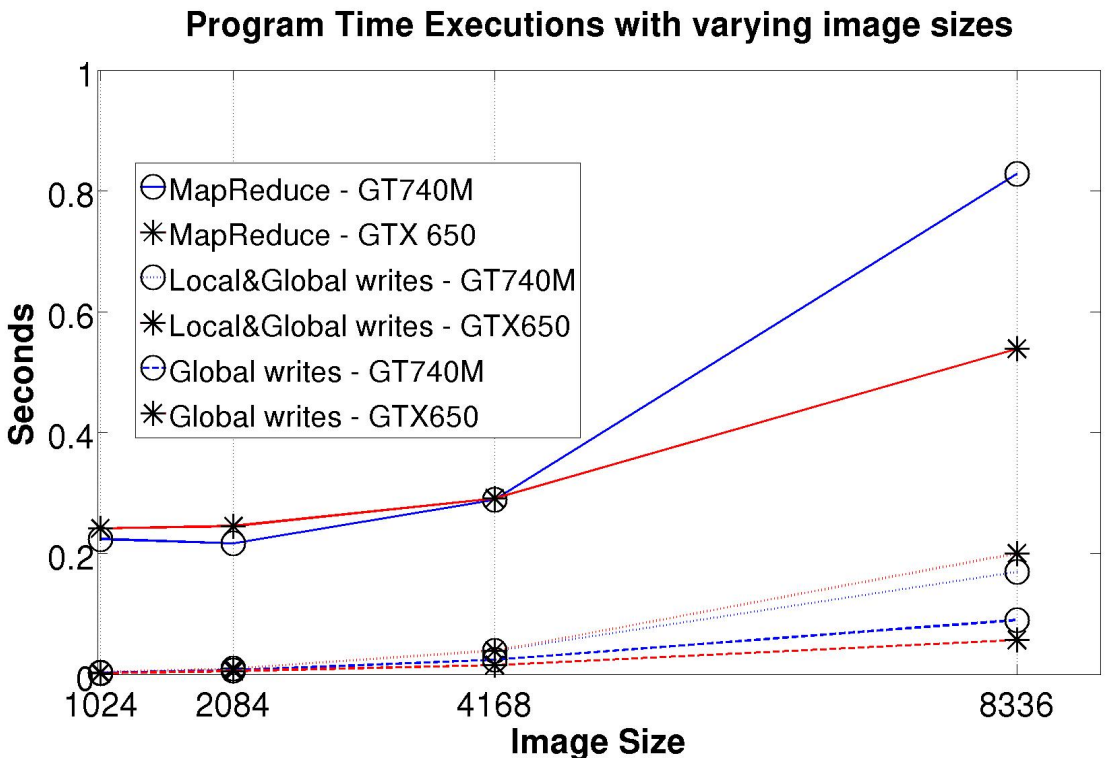


Figure 4.12: Program Time Executions with varying image sizes

5 Conclusions and Future Work

5.1 Conclusion

This report has presented a detailed analysis of both theoretical and implementation aspects of the LBP algorithm. A thorough study of the OpenCL framework was conducted to design, test and analyse various parallel implementation of the algorithm. Specific implementation details of the kernels were shown comparatively to find the optimized performance in performing the LBP algorithm. The parallel execution of these kernels were extensively tested on two GPUs, the NVIDIA GeForce GT740M and GTX650. Results turned out differently than what was expected. We found a significant difference between the program utilising only global memory accesses, to the other two programs relying on both global and local memory space for their operations. With supported information obtained from various profiling tool-kits, atomic operations were found to be behind the huge contrast in performance. The implementation with local and global memory writes was seen taking two to three times longer; while MapReduce took up to seven or eight times longer when a large sized 8336x8336 image was used as input. These kernels can be easily further extended to LBP based feature detection methods used in many applications including medical image processing.

Focus on analysis was then shifted from algorithmic to the physicality of processing devices. The availability of physical resources on each device was not that different but not enough research and profiling were done to justify their individual performance characteristics.

5.2 Future Work

In image processing applications, it is important the feature vector accurately represents the intensity distribution of the relevant areas of the image. The conversion from color image to a gray-level image during the pre-processing step can cause some loss of information. In a parallel implementation, each of the three channels can be processed separately, allowing for individualized processing and optimizations. Generalized LBP algorithms use multiple neighbourhoods around each pixel therefore extensions of the current implementation can be explored. The histogram computation of LBP could be further optimized too.

More specific optimizations and research in this work could be looking into why atomic increments in global memory buffer objects are executed much faster than in a locally defined memory object. A clear understanding of how atomic operations are done is needed.
Tests done on the NVIDIA GTX650 GPU device was on a different machine than the NVIDIA GT740M device therefore running the NVIDIA Compute Visual Profiler for NVIDIA GTX650 was limited in this aspect and time. But doing so could give crucial information about the device and provide crucial information (such as the number of registers used by active work-items) into further optimizing its physical limits.

The second stage of experiments was a late inclusion into the project therefore not much analysis were done between the GPU devices resources and their time performances. Spending more time using the NVIDIA visual profiler profiling the different devices separately certainly will give a more detailed analysis and help provide reasons behind their performance plots.
Because of OpenCL's heterogeneous computing environment, the developed implementation methods could be tested on larger parallel architectures such as the University of Canterbury Blue-Fern supercomputer.

Bibliography

[1] Trygve Randen and John Hakon Husoy. Filtering for texture classification: A comparative study. volume 21, pages 291–310. IEEE, 1999.

[2] Timo Ojala, Matti Pietikäinen, and David Harwood. Performance evaluation of texture measures with classification based on kullback discrimination of distributions. *Pattern recognition*, 1:582–585, 1994.

[3] Timo Ojala, Matti Pietikäinen, and David Harwood. A comparative study of texture measures with classification based on featured distributions. volume 29, pages 51–59. Elsevier, 1996.

[4] Li Wang and Dong-Chen He. Texture classification using texture spectrum. volume 23, pages 905–910. Elsevier, 1990.

[5] Ramin Zabih and John Woodfill. Non-parametric local transforms for computing visual correspondence. In *Computer VisionECCV'94*, pages 151–158. Springer, 1994.

[6] Timo Ojala and Matti Pietikäinen. Unsupervised texture segmentation using feature distributions. volume 32, pages 477–486. Elsevier, 1999.

[7] Mäenpää Topi, Pietikäinen Matti, and Ojala Timo. Texture classification by multi-predicate local binary pattern operators. In *Pattern Recognition, International Conference on*, volume 3, pages 3951–3951. IEEE Computer Society, 2000.

[8] Timo Ojala, Matti Pietikainen, and Topi Maenpaa. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. volume 24, pages 971–987. IEEE, 2002.

[9] Marko Heikkilä, Matti Pietikäinen, and Cordelia Schmid. Description of interest regions with local binary patterns. volume 42, pages 425–436. Elsevier, 2009.

[10] Matti Pietikaeinen, Timo Ojala, Jarkko Nisula, and Jouni Heikkinen. Experiments with two industrial problems using texture classification based on feature distributions. In *Photonics for Industrial Applications*, pages 197–204. International Society for Optics and Photonics, 1994.

[11] Loris Nanni, Alessandra Lumini, and Sheryl Brahnam. Local binary patterns variants as texture descriptors for medical image analysis. volume 49, pages 117–125. Elsevier, 2010.

[12] G Castellano, L Bonilha, LM Li, and F Cendes. Texture analysis of medical images. volume 59, pages 1061–1069. Elsevier, 2004.

[13] Timo Ojala, Kimmo Valkealahti, Erkki Oja, and Matti Pietikäinen. Texture discrimination with multidimensional distributions of signed gray-level differences. volume 34, pages 727–739. Elsevier, 2001.

[14] Mäenpää Topi, Ojala Timo, Pietikäinen Matti, and Soriano Maricor. Robust texture classification by subsets of local binary patterns. In *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, volume 3, pages 935–938. IEEE, 2000.

[15] Matti Pietikäinen, Abdenour Hadid, Guoying Zhao, and Timo Ahonen. *Computer vision using local binary patterns*, volume 40. Springer, 2011.

[16] Kristin J Dana, Bram Van Ginneken, Shree K Nayar, and Jan J Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics (TOG)*, 18(1):1–34, 1999.

[17] NVIDIA Corporation. *NVIDIA CUDA Programming Guide Version 2.3*, 2009.

[18] NVIDIA Corporation. *NVIDIA CUDA Best Practices Guide Version*, 2010.

[19] *The OpenCL Specification*, version 1.0 revision 48 edition, 2010.

[20] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. volume 96, pages 879–899. IEEE, 2008.

[21] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki, and Satoru Tagawa. *The OpenCL Programming Book*, volume 63. 2010.

[22] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[23] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[24] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[25] Narmada Naik and GN Rathna. Real time face detection on gpu using opencl. *Computer Science*, 2014.

[26] CYN Dwith and GN Rathna. Parallel implementation of lbp based face recognition on gpu using opencl. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on*, pages 755–760. IEEE, 2012.

[27] Jeaff Wang and Richard Abrich. Face detection with improved local binary patterns in cuda. 2013.

[28] Gregor Zolynski, Tim Braun, and Karsten Berns. Local binary pattern based texture analysis in real time using a graphics processing unit. *VDIBERICHT*, 2012:321, 2008.

[29] Timo Stich. Opencl on nvidia gpus, 2009.

[30] Matthew Scarpino. *OpenCL in Action: how to accelerate graphics and computation*. Manning, 2012.

[31] NVIDIA Corporation. *NVIDIA CUDA Programming Guide Version 4.2*, 2012.

[32] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, James Fung, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

[33] Nvidia geforce gtx 650 specifications, 2014.

[34] Nvidia geforce gt 740m specifications, 2014.

[35] Nvidia cuda gpu occupancy calculator, 2014.

A OpenCL Kernel Code

Appendix A1 : Kernel Code for LBP Image Generation

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
      CLK_ADDRESS_CLAMP |
      CLK_FILTER_NEAREST;
uint4 read(image2d_t img, int w, int h, int2 coords)
{
  int ix = coords.s0;
  int iy = coords.s1;
  if (ix == -1) { ix = w-1;}
  else if (ix == w) { ix = 0;}
  if (iy == -1) { iy = h-1;}
  else if (iy == h) { iy = 0;}

  return read_imageui(img, sampler, (int2)(ix, iy));
}

uint img_lbp(image2d_t img, image2d_t imgout, int w, int h, int ix, int iy)
{
  uint4 pixelc = read(img, w, h, (int2)(ix, iy)); //Center pixel
  uint4 pixel0 = read(img, w, h, (int2)(ix - 1, iy - 1)); //First pixel neighbour
  uint4 pixel1 = read(img, w, h, (int2)(ix    , iy - 1)); //Second pixel neighbour
  uint4 pixel2 = read(img, w, h, (int2)(ix + 1, iy - 1)); //Third pixel neighbour
  uint4 pixel3 = read(img, w, h, (int2)(ix + 1, iy    )); //Fourth pixel neighbour
  uint4 pixel4 = read(img, w, h, (int2)(ix + 1, iy + 1)); //Fifth pixel neighbour
  uint4 pixel5 = read(img, w, h, (int2)(ix    , iy + 1)); //Sixth pixel neighbour
  uint4 pixel6 = read(img, w, h, (int2)(ix - 1, iy + 1)); //Seventh pixel neighbour
  uint4 pixel7 = read(img, w, h, (int2)(ix - 1, iy    )); //Eighth pixel neighbour
  uint thres0 = 0;
  uint thres1 = 0;
  uint thres2 = 0;
  uint thres3 = 0;
  uint thres4 = 0;
  uint thres5 = 0;
  uint thres6 = 0;
  uint thres7 = 0;

  //Perform Thresholding on neighbours with the center pixel value
```

```
  if (pixel0.s0 >= pixelc.s0) { thres0 = 1; }
  if (pixel1.s0 >= pixelc.s0) { thres1 = 1; }
  if (pixel2.s0 >= pixelc.s0) { thres2 = 1; }
  if (pixel3.s0 >= pixelc.s0) { thres3 = 1; }
  if (pixel4.s0 >= pixelc.s0) { thres4 = 1; }
  if (pixel5.s0 >= pixelc.s0) { thres5 = 1; }
  if (pixel6.s0 >= pixelc.s0) { thres6 = 1; }
  if (pixel7.s0 >= pixelc.s0) { thres7 = 1; }

  //Neighbours thresholded value multiplied by powers of 2, depending on their no. neighbour
  thres0*=1;
  thres1*=2;
  thres2*=2*2;
  thres3*=2*2*2;
  thres4*=2*2*2*2;
  thres5*=2*2*2*2*2;
  thres6*=2*2*2*2*2*2;
  thres7*=2*2*2*2*2*2*2;

  //Summed the neighbour thresholded values into a LBP label for the center pixel
  pixelc.s0 = thres0 + thres1 + thres2 + thres3 + thres4 + thres5 + thres6 + thres7;

  write_imageui(imgout, (int2)(ix, iy), pixelc);

  return pixelc.s0;
}
```

Appendix A2 : Kernel Code for LBP Histogram Generation First Implementation

```
__kernel void histogram_partial_image(read_only image2d_t img, write_only image2d_t imgout
                                      , __global uint* histogram)
{
    int image_width = get_image_width(img);
    int image_height = get_image_height(img);
    int x = get_global_id(0);
    int y = get_global_id(1);

    if ((x < image_width) && (y < image_height)) {
        uint clr = img_lbp( img, imgout, image_width, image_height, x, y);
        atomic_inc(&histogram[clr]);
    }
}
```

Appendix A3 : Kernel Code for LBP Histogram Generation Second Implementation

```
__kernel void histogram_partial_image(read_only image2d_t img, write_only image2d_t imgout
                                      , __global uint* histogram)
{
    int local_size = (int)get_local_size(0) * (int)get_local_size(1);
    int image_width = get_image_width(img);
    int image_height = get_image_height(img);
    int x = get_global_id(0);
    int y = get_global_id(1);
    __local uint tmp_histogram[256];
    int tid = get_local_id(1) * get_local_size(0) + get_local_id(0);
    int j = 256;
    int indx = 0;
    do {
       if (tid < j) {
          tmp_histogram[indx+tid] = 0;
       }
       j -= local_size;
       indx += local_size;
    } while (j > 0);

    barrier(CLK_LOCAL_MEM_FENCE);

    if ((x < image_width) && (y < image_height)) {
       uint clr = img_lbp( img, imgout, image_width, image_height, x, y);
       atomic_inc(&tmp_histogram[clr]);
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    if (local_size >= 256) {
       if (tid < 256){
          atomic_add(&histogram[tid], tmp_histogram[tid]);
       }
    }
    else {
       j = 256;
       indx = 0;
       do {
          if (tid < j) {
             atomic_add(&histogram[indx+tid], tmp_histogram[indx+tid]);
          }
          j -= local_size;
          indx += local_size;
       } while (j > 0);
```

```
    }
}
```

Appendix A4.1 : First Kernel Code for LBP Histogram Generation MapReduce Implementation

```
__kernel void histogram_partial_image(read_only image2d_t img, write_only image2d_t imgout
                                      , __global uint* histogram)
{
    int local_size = (int)get_local_size(0) * (int)get_local_size(1);
    int image_width = get_image_width(img);
    int image_height = get_image_height(img);
    int group_indx = (get_group_id(1) * get_num_groups(0) + get_group_id(0)) * 256;
    int x = get_global_id(0);
    int y = get_global_id(1);
    __local uint tmp_histogram[256];
    int tid = get_local_id(1) * get_local_size(0) + get_local_id(0);
    int j = 256;
    int indx = 0;
    do {
       if (tid < j) {
          tmp_histogram[indx+tid] = 0;
       }
       j -= local_size;
       indx += local_size;
    } while (j > 0);

    barrier(CLK_LOCAL_MEM_FENCE);

    if ((x < image_width) && (y < image_height)) {
       uint clr = img_lbp( img, imgout, image_width, image_height, x, y);
       atomic_inc(&tmp_histogram[clr]);
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    if (local_size >= 256) {
       if (tid < 256){
          histogram[group_indx + tid] = tmp_histogram[tid];
       }
    }
    else {
       j = 256;
       indx = 0;
       do {
          if (tid < j) {
             histogram[group_indx + indx + tid] = tmp_histogram[indx + tid];
```

```
          }
          j -= local_size;
          indx += local_size;
       } while (j > 0);
    }
}
```

Appendix A4.2 : Second Kernel Code for LBP Histogram Generation MapReduce Implementation

```
__kernel void histogram_sum_partial(global uint *partial_histogram,
                                    int num_groups, global uint *histogram)
{
   int tid = (int)get_global_id(0);
   int group_indx;
   int n = num_groups;
   local uint tmp_histogram[256];

   tmp_histogram[tid] = partial_histogram[tid];
   group_indx = 256;
   while (--n > 0) {
      tmp_histogram[tid] += partial_histogram[group_indx + tid];
      group_indx += 256;
   }

   histogram[tid] = tmp_histogram[tid];
}
```